A Twitter Bot Based on A Tableau Solver for GL Logic

Bachelor's Project Thesis

Jeroen van Gelder, s3813053, j.h.van.gelder.1@student.rug.nl,
Supervisor: Prof. Dr. L.C. Verbrugge

Abstract: In 2017 a tableau solver for provability logic (GL) was built, which faced a number of memory issues. In an attempt to improve memory efficiency, a new tableau solver is built for GL using an object-oriented approach in Python, as well as implementing both depth-first search and culling of closed branches. Alongside, a procedural generator for formulas of GL is built, and a Twitter bot is made that tweets all found theorems of GL at an interval of three hours. Depth-first search is implemented in order to find the easy answer the quickest, as a complete tableau is not required. The culling of closed branches is implemented in an attempt to keep memory usage to a minimum. After testing the system with more than one million formulas, no out-of-memory issues are found, indicating that memory efficiency has improved.

1 Literature Review

The provability logic 'GL' is a propositional modal logic that is an extension to the modal logic K (Kripke). The logic's name 'GL' is an abbreviation of 'Gödel-Löb'. This logic is a result of the combination of the works of two logicians: K. Gödel and M.H. Löb.

1.1 Background

Gödel is well known for both his notions about provability and his incompleteness theorems on provability limitations of formal axiomatic systems. In Gödel's famous set of letters, he described the logic behind complex, or seemingly incorrect, self-referencing statements. An example of such a statement is 'I am lying', which mentions its own incorrectness. Such a sentence can, at face value, be neither true nor false, as it contradicts itself in both cases. Gödel observed that a sufficiently strong language or arithmetic, that has the ability of expressing its own provability, can make this puzzling sentence into a proper, logically correct statement. The sentence 'I am lying' can then be reformulated into 'I am unprovable,' referring to its own concept of provability (Smorynski, 2004).

The other influential factor in the shaping of the logic GL was Löb's theorem. In 1955, M.H. Löb wrote a paper titled 'Solution to a Problem of Leon Henkin' in which he answered a question asked by L. Henkin. Henkin posed the question whether anything could be said about self-referencing sentences that say something about their own provability, rather than just their own unprovability, as investigated by Gödel. Löb's answer to this question was: if $PA \vdash Prov(\ulcorner A \urcorner) \rightarrow A$, then $PA \vdash A$, which is now known as Löb's theorem. Löb formalized this theorem as: $PA \vdash Prov(\ulcorner Prov(\ulcorner A \urcorner) \rightarrow A \urcorner) \rightarrow Prov(\ulcorner A \urcorner)$ (Löb, 1955). These expressions are in regards to the formal arithmetic Peano Arithmetic (PA), which will be briefly discussed later.

Combining these two influential points in providing a logical basis for provability, resulted in the manifestation of GL. In GL, the concepts of provability are expressed using the modal box $\Box$ and diamond $\Diamond$ operators. The interpretation of the box operator of modal logic is adapted from 'it is necessary in T that...' to 'it is provable in T that...'. Likewise, the diamond operator would be interpreted as 'it is consistent in T that...', or 'it is not provable in T that not...,' as opposed to 'it is possible in T that...,' where T is a sufficiently strong formal system. The distinguishing element, making GL an extension of the normal modal logic K, is the addition of the formalization of Löb's theorem, expressed using the modal operators: $\Box(\Box A \rightarrow A) \rightarrow \Box A$. This formula does not hold in K (Boolos, 1993).

Gödel and Löb focused their research on provability in formal arithmetical systems. For GL, soundness and completeness are defined with respect to Peano Arithmetic (PA). PA is a formal mathematical system for which a provability predicate Prov(x) can be defined. The function takes the Gödel number of a formula, a numerical representation of a formula denoted as '⌜A⌝'. The function Prov(x) can be read as $\exists p Proof(p, x)$, meaning the Gödel number p codes a correct proof in PA of the formula with Gödel number x. This is similar to how one would interpret $\Box A$ in GL; meaning that A is provable in a sufficiently strong formal system, in this case PA (Artemov, 2001; Verbrugge, 2017).

The satisfiability problem for the logic GL is, similar to e.g. S4, which is known to be complete in P-SPACE. This entails that there is a polynomial p such that the set of theorems of GL is computable in space p with respect to the length of the formula (Ladner, 1977). Note that the polynomial space is the upper bound. P-SPACE complete logics can therefore be quite memory intensive, which is part of the drive behind this research.

1.2 Research Question

In 2017, another student, Tim van Loo, designed a tableau solver in GL (van Loo, 2017). In their paper, it is mentioned that they faced numerous memory issues. Part of the reason for this was that the complete tableau had to persist during solving in order for the solver to display the final tableau to the user. Therefore, we will investigate whether we can build a solver that will avoid these memory limitations by applying both depth-first search, and culling of closed branches. The research question is as follows:

Can memory efficiency be improved of a tableau solver for GL logic when compared to the bachelor's thesis of Van Loo (2017) by applying depth-first search and culling of closed branches?

Our hypothesis is that the combination of depth-first search and branch culling will allow quick termination upon finding one open branch, and reduce memory usage by removing all closed branches from memory.

Alongside the solver, a procedural generator will be designed to provide continuously growing formulas to the solver. In addition, to display the process to the public, a Twitter bot is made which Tweets every found tautology of GL. The Twitter bot can be found at `https://twitter.com/GL_LogicBot`.

2 GL characteristics

This section will delve deeper into the specifics of GL. We will cover the semantics and prove that GL is both transitive and converse well-founded. Aside from that, proofs for the modal soundness and completeness of GL will be provided as well.

2.1 Semantics

As previously stated, GL is a propositional modal logic. It is similar to the modal logic K in the sense that all regular propositional operators are also present in GL, as well as the contradiction operator $\bot$. Although the modal operators are interpreted differently, the distinguishing factor that extends GL from K is the addition of the axiom $\Box(\Box A \to A) \to \Box A$.

The modal logic K is valid in all models $I = \langle W, R, V \rangle$. However, due to the added Löb axiom, GL is not. For GL to be sound and complete, the relations R of GL are both transitive and converse well-founded.

Transitivity entails that for all worlds w_1, w_2 and w_3 in W, if $w_1 R w_2$ and $w_2 R w_3$, then $w_1 R w_3$. As with the logic K4, transitivity causes the formula $\Box A \to \Box\Box A$ to hold (Rautenberg, 1983).

A converse well-founded relation is a relation in which there are no infinite, ascending sequences. An infinite sequence such as $w_1 R w_2 R w_3 R w_4 ...$ is thus not allowed. An added consequence of a relation being converse well-founded is that the relation is irreflexive. A reflexive relation wRw can be interpreted as a never-ending loop, or an infinitely ascending sequence. As stated, this is not allowed, thus the relation is inherently irreflexive (Solovay, 1976; Boolos, 1993; Verbrugge, 2017).

2.2 Semantic proofs

In this section, we provide the proofs of characterizing transitivity for the formula $\Box A \to \Box\Box A$ and of both the characterizing transitivity and converse well-founded constraints for the formula $\Box(\Box A \to A) \to \Box A$. These proofs are inspired by the paper by Solovay (1976) and the book by Boolos (1993).

Transitivity Lemma: For all frames F, $F = \langle W, R\rangle$ satisfies $\Box A \rightarrow \Box\Box A$ iff R is transitive. *Proof:* Assume that R is transitive. Let $w \in W$ where $\Box A$ holds at w. We will now show that $\Box\Box A$ holds at w.

Suppose $\Box\Box A$ does not hold at w. Then for some x, wRx, $\Box A$ does not hold at x. Then for some y, xRy, A does not hold at y. However, due to transitivity and given wRx and xRy, wRy. Since $\Box A$ holds at w, it cannot be that A does not hold at y. Therefore, R must be transitive. Thus, if $\Box A$ holds at w, $\Box\Box A$ also holds at w. Therefore, for transitive R, $F \vdash \Box A \rightarrow \Box\Box A$.

Suppose now that the formula $\Box A \rightarrow \Box\Box A$ holds in the frame F and that wRx and xRy where $w, x, y \in W$. We will show that wRy.

Let v be the valuation on W such that for $z \in W$, $v(A, z) = 1$ iff wRz. Then $v(\Box A, w) = 1$ and $v(\Box\Box A, w) = 1$ due to the assumption that $\Box A \rightarrow \Box\Box A$ holds in F. Therefore, $v(\Box A, x) = 1$ since wRx and $v(A, y) = 1$ since xRy. The latter confirms wRy as $v(\Box A, w) = 1$. Thus if $\Box A \rightarrow \Box\Box A$ holds in all frames $F = \langle W, R\rangle$, iff R is transitive.

Converse Well-Founded Lemma: The frame $F = \langle W, R\rangle$ satisfies $\Box(\Box A \rightarrow A) \rightarrow \Box A$ iff R is both transitive and converse well-founded.

Proof: We start by showing that for a non-empty set X with no R-greatest element, meaning it is infinitely ascending. Let $w \in X$, and let v be the valuation on W such that for every $a \in W$, $v(A, a) = 1$ iff $a \notin X$. We show that $\Box(\Box A \rightarrow A)$ holds in w, thus F does not satisfy $\Box(\Box A \rightarrow A) \rightarrow \Box A$.

Suppose wRx, where $x \in W$. Assume that A does not hold in x, then $x \in X$. For some $y \in X$ and $y \in W$ where A does not hold in y, xRy. This means that $\Box A$ does not hold in x, thus we can safely conclude that $\Box A \rightarrow A$ holds in x and since wRx, $\Box(\Box A \rightarrow A)$ holds in w.

Since $w \in X$, for some $x \in X$, wRx and $x \in W$. This means A does not hold in x, thus $\Box A$ does not hold in w. The combination of this and the fact that $\Box(\Box A \rightarrow A)$ holds in w, contradicts that F satisfies $\Box(\Box A \rightarrow A) \rightarrow \Box A$.

Suppose now that R is both transitive and converse well-founded. Suppose also that for a given valuation $\Box A$ does not hold at w. Then let X be the set of worlds where $x \in X$ iff $x \in W$, wRx and A does not hold in x. Since $\Box A$ does not hold at w, for some y, wRy, A does not hold at y. This means that $y \in X$ by the characteristics of X, thus X is non-empty and by the converse well-founded constraint, for some $x \in X$, xRz for no $z \in X$, else the set has an infinitely ascending relation. Suppose for world z, xRz. Since $\Box A$ holds at w and wRx, by transitivity also wRz, thus A holds at z, confirming $z \notin X$. This also means that $\Box A$ holds at x, thus $\Box A \rightarrow A$ does not hold at x, thus by wRx, $\Box(\Box A \rightarrow A)$ does not hold at w, thus it is safe to conclude $\Box(\Box A \rightarrow A) \rightarrow \Box A$ holds at w. Therefore, for transitive and converse well-founded R, F satisfies $\Box(\Box A \rightarrow A) \rightarrow \Box A$.

2.3 Tableau Rules

We will now provide the tableau rules, specific to GL. From the chapter written by Goré (1998) and the article by Rautenberg (1983), we find and derive the tableau rules of GL for the modal operators.

For the box rule and the derived negated diamond rule, the interpretation is similar to the interpretation in K. It is important to note that from $\Box P$ we get $\Box P$ in the reachable world as well. This is an abbreviation of the transitivity constraint. Additionally, note that from $\Diamond P$, $\neg\Diamond P$ is introduced in the reachable world. This is an abbreviation of the converse well-founded constraint, prohibiting infinitely ascending relations. The tableaux are as follows:

1.	$\Box P, i$	1.	$\neg\Diamond P, i$
2.	irj	2.	irj
3.	$\mid$	3.	$\mid$
4.	$\Box P, j$	4.	$\neg\Diamond P, j$
5.	P, j	5.	$\neg P, j$

Note that these rule are applied to each world j on the branch, including worlds introduced after the initial application of the box rule.

The negation of the box rule and diamond rule are quite different from K due to the Löb axiom that defines GL. It still holds that

$\Box P \iff \neg\Diamond\neg P$. However, if we rewrite the Löb axiom, we get the following (Goré, 1998):

$$\Box(\Box A \to A) \to \Box A$$

$$\neg\Box A \to \Box(\Box A \to A)$$

$$\Diamond\neg A \to \Diamond(\Box A \wedge \neg A)$$

Therefore, the tableau rules are as follows:

1.	$\neg\Box P, i$	1.	$\Diamond P, i$
2.	$\vert$	2.	$\vert$
3.	irj	3.	irj
4.	$\Box P, j$	4.	$\neg\Diamond P, j$
5.	$\neg P, j$	5.	P, j

Note that the world j is introduced as a new world to the branch.

Additionally, the transitivity rule (τ) is defined as follows:

1.	irj
2.	jrk
3.	$\vert$
4.	irk

2.4 Soundness and Completeness

In this section, proofs are provided for the modal soundness and completeness of GL, using methods described in the book by Priest (Priest, 2001). GL has also been proven to be arithmetically complete with PA, which means that $GL \vdash A$ if and only if for all interpretations f, $PA \vdash f(A)$. This means that every formula that is valid in PA is a theorem of GL (Solovay, 1976).

2.4.1 Modal Soundness

The proof for modal soundness is an adaptation of the proof of soundness of basic modal logic (Priest, 2001).

Definition: Let $I = \langle W, R, v\rangle$ be any GL interpretation and let b be any branch of the tableau. Then I is faithful to b iff there is a map, f, from the natural numbers to W such that:

> For every node A, i on b, A is true at $f(i)$ in I.
> If irj is on b, $f(i)Rf(j)$ in I.

We say that f shows I to be faithful to b.

Soundness Lemma: Let b be any branch of a tableau, and $I = \langle W, R, v\rangle$ be any transitive, converse well-founded interpretation. If I is faithful to b, and a tableau rule is applied to it, then it produces at least one extension b', such that I is faithful to b'.
Proof: Let f be a function which shows I to be faithful to b. The proof will be constructed in a case-by-case manner for both the box $\Box$ and diamond $\Diamond$ rules, including their negations, as well as the transitivity rule. We will show that the the branches b' resulting from the GL tableau rules are faithful to I. The regular propositional operators are excluded in the proof due to them being similar to K, which is both sound and complete.

Suppose $\Box A, i$ occurs on b and we apply the GL rule as defined in Section 2.3. Since b is faithful to I, we know that $\Box A, i$ is true at $f(i)$. Then for any irj on b, where $f(i)Rf(j)$ holds, we have both $\Box A, j$ and A, j on an extension b'. Since $\Box A$ is true at $f(i)$, we can conclude from transitivity of I that $\Box A$ is also true at $f(j)$ and by definition of the $\Box$-rule, we can conclude that A is also true at $f(j)$. Therefore, I is faithful to the extension b' of the branch.

Suppose $\neg\Diamond A, i$ occurs on b. We can then rewrite this through the normal modal rule for $\neg\Diamond$ and extend the branch with $\Box\neg A, i$. Since I is faithful to b, $\neg\Diamond A$ is true at $f(i)$. Hence, $\Box\neg A$ is true at $f(i)$. Thus I is faithful to the extension of b.

Suppose $\Diamond A, i$ occurs on b and we apply the GL rule for $\Diamond$ as defined in Section 2.3. The rule yields the nodes irj, $\neg\Diamond A, j$ and A, j. Since b is faithful to I, $\Diamond A$ is true at $f(i)$. Therefore, for some $w \in W$, $f(i)Rw$ and A is true at w. By converse well-foundedness of I, this is the last such w. Let f' be the same as f, except that $f'(j) = w$. Note that f' also shows that I is faithful to b, since f and f' differ only at j, which does not occur on b. By definition, $f'(i)Rf'(j)$ and A is true at $f'(j)$. Also the sentence $\neg\Diamond A$ is true at $f'(j)$. Therefore, f' shows I to be faithful to the extended branch.

Similar to the negation of the $\Diamond$-rule, suppose $\neg\Box A, i$ occurs on b. We can the apply the normal modal rule for $\neg\Box$ and extend the branch with the sentence $\Diamond\neg A, i$. I is faithful to b, thus $\neg\Box A$ is true at $f(i)$. Therefore, $\Diamond\neg A$ is true at $f(i)$ and I is faithful to the extension of b.

Now suppose that both irj and jrk occur on b. Since I is faithful to b, $f(i)Rf(j)$ and $f(j)Rf(k)$. Due to R being transitive as is required by GL, $f(i)Rf(k)$. Thus I is faithful to the extension of b.

Soundness Theorem: If $GL \vdash A$ then $GL \vDash A$.

Proof: Suppose $GL \nvDash A$. Then there is an interpretation $I = \langle W, R, v\rangle$, which is transitive and converse well-founded, that makes every premise true, but A false in some world w. Let f be any function such that $f(0) = w$, which shows I to be faithful to the initial list. By repeated application of the Soundness Lemma, we continuously find at least one extension to which I is faithful, resulting in a whole branch b such that I is faithful to every initial section of it. Thus I is faithful to b itself. If b were to be a closed branch, there must be some contradiction within the branch, which is not possible since I is faithful to b. Therefore, the tableau is open, or $GL \nvdash A$.

2.4.2 Modal Completeness

The proof for modal completeness is also an adaptation of the proof of completeness of basic modal logic (Priest, 2001).

Definition: Let b be an open branch of a tableau and $I = \langle W, R, v\rangle$ be the interpretation induced by b as follows: $W = \{w_i : i \textit{ occurs on } b\}$. w_iRw_j iff irj occurs on b. If p, i occurs on b, then $v_{w_i}(p) = 1$; if $\neg p, i$ occurs on b, then $v_{w_i}(p) = 0$. Otherwise $v_{w_i}(p)$ is arbitrary.

Completeness Lemma: Let b be any open complete branch of a tableau. Let $I = \langle W, R, v\rangle$ be the interpretation induced by b. Then for all formulas A:

if A, i is on b then A is true at w_i
if $\neg A, i$ is on b then A is false at w_i

Moreover, R is both transitive and converse well-founded.

Proof: We will prove the completeness lemma by induction on the complexity of A.

Base step: If sentence A is atomic, the result is true.

Inductive Hypothesis: Suppose we have two arbitrary formulas B and C where:

if B, i is on b then B is true at w_i
if $\neg B, i$ is on b then $\neg B$ is true at w_i
if C, i is on b then C is true at w_i
if $\neg C, i$ is on b then $\neg C$ is true at w_i

Inductive Step: We will only discuss the modal operators as these differentiate GL from the normal modal logic K. The logic K is proven to be both sound and complete (Priest, 2001).

Suppose that A is of the form $\Box B$. If $\Box B, i$ is on b, then for all j such that irj is on b, $\Box B, j$ and B, j are on b. Thus for all w_j such that w_iRw_j, $\Box B$ and B are true at w_j by inductive hypothesis. Therefore, $\Box B$ is true at w_i, as is required. There is, however, a situation where the application of the $\Box$-rule could also cause a new world to be introduced. If $B = \Diamond C$, then we rewrite $\Box B, i$ as $\Box\Diamond C, i$. This means that for all w_j such that w_iRw_j, $\Box\Diamond C, j$ and $\Diamond C, j$ are on b, and with the $\Diamond$-rule introducing a new world, this could end in an infinite branch. From Section 2.3 we observe that applying the rule for $\Diamond C, j$ results in a new world w_k such that w_jRw_k, as well as a formula $\neg\Diamond C, k$ and C, k. The formula $\Box\Diamond C, j$ results in $\Box\Diamond C, k$ and $\Diamond C, k$. The latter contradicts with its negation resulting from the diamond rule, hence the branch will close and there will be no infinite branch.

The same holds for the situation where A is in the form $\neg\Diamond B$, since $\neg\Diamond B \Leftrightarrow \Box\neg B$.

Suppose that A is of the form $\Diamond B$. If $\Diamond B, i$ is on b, then for some j, irj, $\neg\Diamond B, j$ and B, j are on b. By inductive hypothesis, w_iRw_j, $\neg\Diamond B$ and B are true at w_j. Therefore, $\Diamond B$ is true at w_j as is required. However, as with the $\Box$-rule, we must also investigate whether a formula that introduces another new world could cause an infinite branch. For the formula $\Diamond B, i$ to possibly introduce infinitely many new worlds, B must be infinitely long itself, i.e.: $B = \Diamond\Diamond\Diamond...\Diamond C$. This is not possible, hence there will be no infinite branch.

The same holds for the situation where A is in the form $\neg\Box B$, since $\neg\Box B \Leftrightarrow \Diamond\neg B$.

Although the absence of infinite branches proves converse well-foundedness, we must still prove transitivity. This is done with the τ-rule. Given $w_i, w_j, w_k \in W$, suppose that w_iRw_j and w_jRw_k Then both irj and jrk occur on branch b. By the τ-rule of Section 2.3, irk then also occurs on b.

Therefore, $w_i R w_k$ as required for transitivity.

Completeness Theorem: If $GL \models A$ then $GL \vdash A$.

Proof: Suppose that $GL \nvdash A$. Given an open complete branch b of the tableau, take $I = \langle W, R, I \rangle$ to be the induced interpretation. Then by the completeness lemma, not only $GL \nvDash A$, but also the countermodel I is of the right kind: transitive and converse well-founded, both by the completeness lemma.

3 Design Choices

There are two major components to this project: an algorithm that procedurally generates formulas in GL and a tableau solver that checks the validity of said formulas. In addition, an algorithm responsible for tweeting theorems of GL is implemented. For the design choices we will discuss each component separately. (Source code available at `https://github.com/JHvanG/GL-tableaux-solver`)

3.1 Generator

For the generator, there are a few important elements that must be considered. Firstly, the generator must know the syntax of GL and its representation within the system. Secondly, we want each generated formula to be unique. The reasoning behind this is that the aim of this system is to produce unique tautologies in GL, which are then supplied to the Twitter bot. The process of validity checking will be rather resource intensive, as was encountered in the bachelor's thesis by van Loo (2017), hence duplication of formulas should be avoided. To obtain unique formulas, we will adhere to a specific structure of building the formulas.

The chosen structure for building formulas relies on the reusability of older formulas. The generation of the formulas is based on a complexity constraint, which limits the number of connectives in the formulas. The complexity constraint is incremented by one as soon as all formulas within a complexity level are completed and all these formulas are stored. For the complexity of 0, only the atoms 'A', 'B' and '$\perp$' are generated and stored. We have opted to include two atoms and the contradiction due to the fact that we aim to compare with the bachelor thesis by van Loo (2017), in which the most complex formulas consist of combinations of all connectives and these three atoms. For complexity 1 each connective will be included once. For the binary connectives all valid, unique combinations of the atoms from complexity 0 are used.

Any complexity higher than 1 will apply all unary connectives to the complexity below it. In addition, the generator iteratively covers all possible combinations of the complexities below it. For example, a complexity of 2 will cover the combination of the formulas with complexity 0 and 1, as well as the combination of the formulas with complexity 1 with themselves. In other words, for complexity N, the combinations are $[0, N-1]$ up to $[N-1, N-1]$. In this manner, the syntax of GL will automatically be adhered to through the structure of the code. It is important to remember the equivalence between e.g. $A \wedge B$ and $B \wedge A$, and the non-equivalence between $A \rightarrow B$ and $B \rightarrow A$. Note that our algorithm does include all combinations with its own complexity, thus both $A \wedge B$ and $B \wedge A$ in complexity 1. This decision is made to keep the structure of the generation algorithm clean and simple. However, for combinations with a different complexity, only the implications are generated both ways. This is done in an attempt to keep the number of logically equivalent formulas down.

We have opted for an object-oriented approach to the problem. This allows each connective to be its own instance of its respective subclass from the main formula class. The main formula class will contain all parameters and methods that are required for every type of connective. The constructor of the individual connectives controls the proper setting of the parameters, and any connective-specific methods will be included in the subclasses. All connectives are set to be the size of one character so that the length of the formulas can be correctly monitored:

- $\neg$ as `~`
- $\wedge$ as `&`
- $\vee$ as `|`
- $\rightarrow$ as `>`
- $\leftrightarrow$ as `=`
- $\Box$ as `+`
- $\Diamond$ as `-`
- $\perp$ as `#`

Naturally we cannot run the generator infinitely.

For one, because a point will be reached where the formulas are simply be too long for the solver to handle them. But more importantly, since part of this thesis is to build a Twitter bot to post all tautologies to, the length will eventually be limited by the maximum character count for a Tweet. This limit is 280 characters, hence this limit will be adhered to for the maximum formula length, including all brackets.

As mentioned above, the program will be object-oriented. For this reason, the decision was made to design the program in Python 3. Aside from the simple implementation of object-oriented principles, Python offers a rather simplistic development environment and has a wide range of libraries and tools available. This allows for a quick process, simple code and, at the end, straightforward implementation of a Twitter bot.

3.2 Solver

The solver is the main algorithm around which this project revolves. It is designed to determine the validity of a provided formula in GL, which is produced by the generator. As a reminder, each formula is solved by inputting the negated formula and attempting to close all branches. To achieve this, all tableau rules for GL must be known to the algorithm. As stated above, each connective is represented as its own subclass of the formula class. Within these classes, two methods are implemented which apply the tableau rule for the connective, and the tableau rule for its negation. The rules for the negations are called by the rule-application method of the negation class.

The choice is made to include each rule, instead of running a simplification algorithm to rewrite a formula to be, for example, in negated normal form (NNF). This decision is made because both from the bachelor's thesis by van Loo (2017) and research by Goré and Kelly (2007) into tableaux solvers for GL, it was concluded that there was no consistent advantage to rewriting each formula to either be in NNF, to have only $\neg$, $\Box$ or $\vee$, or other manners of rewriting.

Recall the research question for this thesis: *Can memory efficiency be improved of a tableau solver for GL logic when compared to the bachelor's thesis of Tim van Loo (2017) by applying depth-first search and culling of closed branches?*

When investigating the research by van Loo (2017), it becomes clear that the memory issues originated from the fact that the aim was to display the entire search tree of a valid formula in GL, solved as a human would. This could, in some cases, result in excess to 9 million nodes and 400 thousand worlds, or even in RAM issues. Therefore, the concept to be tested is whether a solver without the need for an output proof tree can avoid the memory issues. An example of such a GL tableau solver where there were no memory issues reported can be found in a report by Goré and Kelly (2007), where different approaches to rewriting rules and reducing the number of rules could improve processing speed. The aim for this research, however, is to simply keep all original rules and apply heuristics to improve memory efficiency: depth-first search and culling of closed branches.

Depth-first search is chosen due to its memory efficiency when compared to breadth-first search, in the situation where an early open branch is found. Since we are interested in solving the formula as fast possible, we do not need to fully expand out other branches when we find an open, complete branch. A single open branch already indicates that the formula is not valid in GL, hence no further branches are required.

When a branch closes, depth-first search does not yield a real advantage due to the need of investigating the remaining branches in an attempt to find an open branch. In such situations, we can trim away each closed branch to conserve valuable memory. Again, because we have no interest in conserving the entire tableau.

To keep track of the branch of interest, a Python list will be used in which the formulas on the branch are stored. It is crucial that each sub-branch is aware of all rules that are left to be applied from the branches above it. The reason for this is that with depth-first search, not all rules will be applied to all branches concurrently. To mitigate this, the branches are added as lists within the list representing the current branch. Once the solver reaches the branch and prepares to enter it, all unapplied formulas are copied over to the new branch. This copy must be a deep copy to avoid the possibility of having a reference to the same object in multiple sub-branches, instead of the actual object, and having an edit on one branch be transferred to all other branches. To conserve some memory with the deep copies, we remove the data from the current branch when it is copied over to a single remaining sub-branch. Due to the converse well-founded constraint of GL, infinite branches do not exist, although there might be extreme cases where a tableau will exceed Python's memory limit.

The aim of the solver is to close branches as quickly as possible. Therefore, checking for contradictions, as well as the ordering of the branch is crucial. Each time a new rule is applied, a check is made whether the resulting formulas are already present on the branch, and whether there is a contradiction. The check for duplicates is done to conserve memory. If a contradiction is found, either within the rules to be applied, or within the set of already applied rules, the branch immediately closes.

For the ordering, the goal was to avoid splitting the branch for as long as possible. To achieve this, the different formula classes are sorted into ranks, where the lower ranks occur earlier on the branch. The following ranking system was chosen:

Rank 0:	$\bot$
Rank 1:	$\wedge, \neg\vee, \neg\rightarrow, \neg\neg, \neg\Box, \neg\Diamond$
Rank 2:	$\Box$
Rank 3:	$\Diamond$
Rank 4:	Branch
Rank 5:	$\vee, \neg\wedge, \rightarrow, \leftrightarrow, \neg\leftrightarrow$
Rank 6:	Atoms, $\neg$ Atoms
Rank 7:	$\Box$*, applied to all relations*

This ranking system means that a contradiction ($\bot$), if present, will immediately close the branch. Next all rules not involving branching or worlds are applied, as they are considered to be more easy for the system. Note that for each rule, the shortcut for its negation is directly applied, except for the $\Box$- and $\Diamond$-rules. Next, the $\Box$-rule is applied to all available relations. This is done before the $\Diamond$-rule is applied, since this will introduce new worlds and relations. Once a $\Box$-rule is applied, it is marked as *applied to all relations*, meaning it will not be applied again, until a new relation is added to the branch. The next rules are the branches, if present. These are situated ahead of all other branching functions to prevent a possible situation where the system could add more than two branches, violating the structure of a tableau. Last in the order are the atoms, negated atoms and applied $\Box$-rules. If rules from this rank are encountered, the system knows all possible rules have been applied and the branch is open and complete.

As mentioned above, for all rules, except the $\Box$- and $\Diamond$-rules, the methods applying the negations of the rules are applied immediately. For example, the negation of the disjunction will introduce both formulas in the disjunction as negations directly to the branch. The reasoning behind not applying the shortcuts to $\neg\Box$ and $\neg\Diamond$ is that the negation of the $\Box$-rule would immediately introduce a new relation to the tableau, which we are actively trying to avoid by the aforementioned ordering. It is therefore much simpler to merely introduce the $\Diamond$ and reorder the branch.

3.3 Twitter bot

The Twitter bot is designed using the *Tweepy* library for Python. Every time the solver finds a tautology, it passes the formula over to the Twitter bot. The Twitter bot is set to tweet new tautologies at an interval of three hours. If a new tautology is received before the three hours have passed, the system sleeps. As soon as the system tweets a new tautology, the bot resets its timer and the system goes on to find a new tautology. This leaves the system with three hours to find a new tautology, which should be plenty. If the system were to surpass the three hour mark, the new tautology will be tweeted immediately. In this case, the next tweet will be made three hours from the last tweet. The tweeter runs indefinitely as a background process on a Raspberry Pi 4.

4 Experiments

As mentioned before, the main aim of this research is to investigate the memory efficiency of our GL tableau solver. To achieve this, the *tracemalloc* library for Python is used. This library traces all memory allocations (in bytes) for a specified section of code and returns the peak memory usage. The peak memory usage is measured to determine the maximum strain of the solver on the system. We start the tracing of memory allocation from the moment a new formula is presented to the solver, until the moment the solver is done.

In addition to measuring memory usage, we keep track of the processing time using the Python library *timeit*. Although computation time is rather system dependent, it is useful for comparison within the system. Since the generator always produces formulas in the same order, we measure time and memory usage individually on separate trials to avoid one measurement influencing the other.

The system on which the experiments are performed runs on Windows 10, has 8GB of RAM and has a quad-core 2.80 GHz processor. Although the system tweets each found tautology to its own Twitter page, this is not done during data gathering. This is done to avoid delays caused by the Twitter API.

To determine the correctness of the system, both targeted tests on the solver, and a complete system test were performed.

4.1 Solver test

For the isolated tests on the solver, a variety of formulas was provided to the algorithm. First, both a test of correct application of the transitivity and converse well-founded constraints are tested. These constraints are tested with the formulas $\Box A \to \Box\Box A$ and $\Box(\Box A \to A) \to \Box A$.

Alongside the two characterising formulas for GL, a number of small formulas are presented to the solver. In these formulas, we use all connectives to test whether they function correctly. These formulas are handpicked such that they are solvable by hand to verify the steps taken by the solver. These formulas can be found in Table A1 of the Appendix.

Alongside the handpicked formulas, the set of formulas in the appendix of the thesis by van Loo (2017) is used as well, cross-referencing the validity of the formulas. This is a set of eight formula's. Note that the eighth formula in this set contains atoms that start out at different worlds. Our system is unable to handle such atoms, hence this formula is replaced by a larger, more complex formula. This formula contains more than 146 characters, brackets included. Additionally, the formula contains more conjunctions, which will cause the tableau to branch when negated. Splitting a large tableau more often means more memory consumption, which will test the memory efficiency of the system. The full set of test formulas can be found in Table A2 of the Appendix.

4.2 System test

For the complete system test, the generator and solver are run together. Each time the generator finds a new formula, the formula is handed to the solver which then determines the validity of the formula. The system produces and evaluates formulas at a rather high rate of speed: running the system for 5 minutes yielded almost 40000 validated tautologies. To keep the number of data points controllable, the decision is made to only store the memory and time data for the tautologies. An added benefit of this decision is that since there are less tautologies than invalid formulas in GL, the data covers a larger range of formula lengths within the same time. To gather the data, the system is run for half an hour, both for the memory measurement and the time measurement.

5 Results

With all tests completed, we can now analyze the performance of the solver and the complete system. As mentioned in the Experiments section, the solver was tested on a handpicked set of formulas (Table A1 in the Appendix) and a set of formulas used to test the system of van Loo (2017) (Table A2 of the Appendix).

First and foremost, the transitivity formula $\Box A \to \Box\Box A$ and the GL formula $\Box(\Box A \to A) \to \Box A$ are both evaluated correctly as tautologies. This gives a first indication that the logic and its constraints are modeled correctly. The solving time for both formulas is almost identical, with the transitivity formula taking $43.5\mu s$ and the GL formula taking $46.8\mu s$. The solver uses 23.048kB for the transitivity formula, and 36.336kB for the GL formula.

Aside from these two formulas, the other formulas in the handpicked test set are also evaluated correctly. This was determined by checking the evaluation of the solver with the full tableau of each formula. Within this first test set, the highest peak memory usage was a total of 40.36kB for the formula $\Box\Box\Diamond(A \leftrightarrow A) \wedge \Box\Box\Diamond\Box\Diamond(A \wedge \neg A)$. This formula also had the highest solving time of 1.23ms and was correctly evaluated as invalid.

The lowest peak memory usage was measured for the formula $\neg\bot$, with the peak at 5.96kB. This formula was not solved in the shortest time, as it took the solver 0.205ms. The shortest solving time was $41.2\mu s$, for the formula $A \to A$. This suggests that the solver is almost five times slower in solving the least memory intensive formula, compared to $A \to A$.

However, as stated before, the time measure is not entirely accurate, meaning the measurements not only vary from system to system, but also between tests on the same formula. Therefore, small differences should not be taken as being the full truth. On the other hand, memory usage is independent of variance within the system on which the solver is tested. This means the memory measurements are consistent across multiple tests with the same formulas, making for a more accurate data source for comparisons of formulas. The full results of the tests are included in Table A1 of the Appendix.

For the set of formulas that were also included in the thesis of van Loo (2017), all evaluations are correct. The eighth formula in the test set is a theorem of GL, and is a disjunction of the sixth and the seventh formula, where a number of disjunctions of the seventh formula are replaced with conjunctions for added complexity. The formula was specifically added to investigate what would happen to the solver on a very memory-intensive formula. The formula resulted in no answer within reasonable time. However, the formula also did not produce any out-of-memory errors. In fact it did not produce an error at all. Instead, the system was left running for more than half an hour, continuously traveling further down the tableau, but not at sufficient speed to find the answer within reasonable time.

The other seven formulas, which also produced no memory issues in the thesis of van Loo (2017), all produce their answer in less than a half a second. The slowest evaluation time was 0.292s. This was for the formula $\Box(A \leftrightarrow (\Box(A \vee \Box\bot) \to \Box(A \to \Box\bot))) \to \Box(A \leftrightarrow (\Box\Box\Box\bot \to \Box\Box\bot))$, which also had the highest peak memory usage of 963.852kB.

The time and memory results of the complete system test are shown in Figure 5.1 and 5.2, respectively. Both time and memory are plotted against the formula length in characters, including brackets. As mentioned before, the data is gathered over half an hour of running the system. Due to memory tracing slowing down the program, a total of

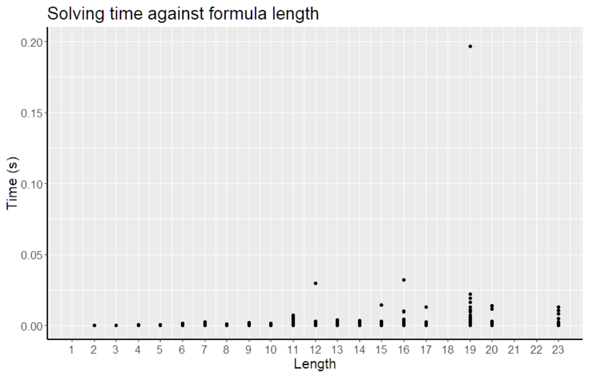

Figure 5.1: This scatter plot shows the computation time of the solver in seconds, plotted against the formula length in characters.

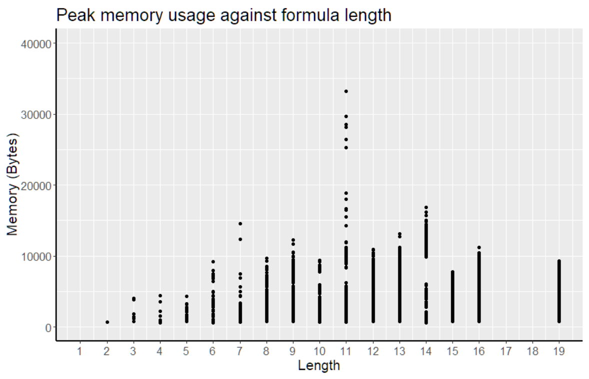

Figure 5.2: This scatter plot shows the peak memory usage of the solver in Bytes, plotted against the formula length in characters.

$533,239$ tautologies are found for the memory data. The time data has $735,057$ tautologies.

Figure 5.1 shows that the majority of tautologies are evaluated within 0.025s. The mean solving time is 0.0003172134s and the median is 0.0002249s. Two formulas of length 12 and 16 take slightly longer than median, with 0.0296812s and 0.0321263s respectively. The formula of length 12 is $\bot \vee (\Box A \vee (B \leftrightarrow B))$ and the formula of length 16 is $((B \leftrightarrow B) \vee \Diamond A) \to (A \leftrightarrow A)$. The mean is influenced heavily by the apparent outlier at a length of 19 characters, where solving the formula $(B \vee A) \vee ((A \wedge B) \vee (\bot \to \bot))$ takes 0.1967872s. The peak memory usage for solving this formula was 6.44kB. We can observe a slight upward trend in computation time as length increases. The higher lengths have less data points due to the data gathering stopping after half an hour.

Figure 5.2 shows that the distribution of the peak memory usage against the formula length is somewhat different from the computation time distribution. It should be noted that, again, the data is gathered over half an hour, which means that the longer formulas that are generated later on are not all in the data set. Moreover, the memory gathering slows down the system, meaning that the longer formulas that are present in the data set of the solving time are not present in the memory data set.

From the figure, we can observe that for the majority of tautologies, the solver has a peak memory usage of less than 20kB. The mean is 2.27kB and the median is 1.256kB. This indicates that the mean is influenced by the set of high peak memory values at length 11. The highest memory usage is for the formula $\Box((A \leftrightarrow A) \vee \Box A)$, which uses 33.2kB of memory at its peak. Solving this formula took 0.0042849s.

6 Conclusion

Recall the research question: *Can memory efficiency be improved of a tableau solver for GL logic when compared to the bachelor's thesis of Tim van Loo (2017) by applying depth-first search and culling of closed branches?* The hypothesis was that by applying depth-first search and culling of closed branches, memory usage is reduced due to simple solutions being found faster and closed branches being removed from memory.

From the results, we can conclude that we did indeed improve memory efficiency. Although a direct memory usage comparison was not possible, the fact that in both the solver test set (Appendix Table A1 and A2) and the complete system test no out-of-memory errors were encountered, suggests that memory efficiency has improved compared to the system of van Loo (2017).

As mentioned before, both the time and peak memory usage plots show an upward trend as formula length increases. The trend for the peak memory usage is much less clear, especially as formula length increases. We must consider the fact that there are approximately $200,000$ tautologies more in the time data set when compared to the memory data set. Those formulas are of a larger length, hence the larger formulas are somewhat underrepresented in the memory usage plot. Given this, we can conclude that a larger memory consumption indicates a longer processing time, although the relation is not one-to-one.

The last formula in the Appendix in Table A2, which the system was unable to solve in reasonable time, consolidates the conclusion that memory issues have been resolved. The formula has been made complex on purpose, adding additional conjunctions which forces branching of the system when negated. Branching increases the memory usage, as unapplied rules have to be copied over to both branches, doubling the space required. The fact that within half an hour, the system did not produce any out-of-memory errors, suggests that the memory issues have been resolved.

However, the very long solving time does indicate that the memory issue is replaced by a slower program. This trade-off will not be troublesome, as it is plenty sufficient for the final goal of the system. With the majority of the first $700,000$ tautologies each being solved within 0.025s, thus more than a million formulas in GL being evaluated within half an hour of running the system, the slower solving time for large formulas will not be noticed. Especially considering that the theorems of GL are tweeted at an interval of three hours, meaning the set of tautologies after running the system for half an hour provides us with enough tautologies for more than 200 years to come.

7 Discussion

Before delving into the discussion, it should be noted that in the current solver the τ-rule is (unnecessarily) applied. This might have helped in improving the speed of solving some formulas, and it could make adaptation to a different transitive modal logic slightly easier. However, it might have also increased computation time by postponing the finding of a contradiction due to investigating more worlds. This is something that is worth investigating for future research.

Reflecting on the process of developing the tableau solver, there are some points of deliberation. Firstly, the method of storing branches can be subject for discussion. Secondly, the question can be asked whether an object-oriented approach is necessary or suited.

The current method used for storing branches utilises Python lists. This method has the advantage of having access to the basic list operations, as well as list comprehension methods in Python. Alternatively, the branches could be represented as instances of a branch object, or even as a node object as done in the thesis of van Loo (2017). This implementation would make the system more object-oriented as a whole, but requires the basic list methods to be represented as functions in the object. It does have the advantage of being able to move some of the custom list operations, such as ordering, to the respective object. This allows for a cleaner separation of functionality within the code. Representation of the tableau will also be simplified, as sub-branches will no longer be present among other formulas, and checking for them is easier.

Opposing this, one could beg the question whether an object-oriented approach is necessary. In light of memory efficiency, it might be more efficient to not have to instantiate every connective as an object. This is very much possible, but would mean that each tableau rule for every connective has to be stored as a separate function, called based on what character is encountered. One of the major challenges with a non-object-oriented approach is how to represent nested connectives. This means that, although it would be an interesting comparison, representational difficulties are likely higher with non-object-oriented approaches.

A possible issue for the complete system is a situation in which it encounters a formula such as the eighth formula in Table A1 of the Appendix. This formula is not solvable within reasonable time. In the current situation, the formula is encountered during testing, rather than during running of the final system. Given the large number of tautologies, the system will not encounter such a formula realistically. However, in the event that it does, it currently keeps running. To prevent this, the system can be stopped prematurely, e.g. by a set time limit. The disadvantage of such a predefined limit is that with larger formulas, more complex tableaux arise as well. These naturally have to be processed for longer, as visible from the time data (Figure 5.1). Therefore, it could be interesting to investigate whether one could base the time limit of computation on either the formula length, or the formula complexity.

Another interesting comparison could be with a similar system in a different language. Python is well-known as not being efficient, both with respect to time and with respect to memory. The reason for this is that Python is an interpreted language, rather than a compiled language. Although slower, this makes Python functions generally shorter and this simplifies things for the programmer. These points, combined with Python experience determined the choice of language for this thesis. However, implementing the system in a language such as Java, C++, or even C, could make for a faster system. A performance comparison could be interesting, especially for formulas of higher complexity.

References

S.N. Artemov. Explicit provability and constructive semantics. *The Bulletin of Symbolic Logic*, 7(1): 1–36, 2001.

G. Boolos. *The Logic of Provability.* Cambridge University Press, 1993.

R. Goré. Tableau methods for modal and temporal logics. In M.D. Agostino, D. Gabbay, R. Hähne, and J. Posegga, editors, *Handbook of Tableau Methods.* Kluwer, Dordrecht, 1998.

R. Goré and J. Kelly. Automated proofsearch in Gödel-Löb provability logic. In *British Logic Colloquium 2007*, 2007.

R.E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.

M.H. Löb. Solution of a problem of Leon Henkin. *The Journal of Symbolic Logic*, 20(2):115–118, 1955.

G. Priest. *An Introduction to Non-Classical Logic.* Cambridge University Press, 2001.

W. Rautenberg. Modal tableau calculi and interpolation. *Journal of Philosophical Logic*, 12:403–423, 1983.

C. Smorynski. Modal logic and self-reference. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd Edition*, volume 11, pages 1–53. Springer Science + Business Media, B.V., 2004.

R.M. Solovay. Provability interpretations of modal logic. *Israel Journal of Mathematics*, 25:287–304, 1976.

Tim van Loo. A tableau prover for GL provability logic. BSc thesis AI, University of Groningen, 2017.

Rineke (L.C.) Verbrugge. Provability Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2017 edition, 2017.

A Appendix

A.1 Test set 1

Formula	Evaluation	Memory (Bytes)	Time (seconds)
$\Box A \to \Box\Box A$	valid	23048	0.0000435
$\Box(\Box A \to A) \to \Box A$	valid	36336	0.0000468
$\neg A \lor A$	valid	6696	0.000415
$A \to A$	valid	6864	0.0000412
$\neg\bot$	valid	5960	0.000205
$(A \land B) \lor (\neg A \lor \neg B)$	valid	7648	0.00108
$\Diamond A \to \neg\bot$	valid	6640	0.0000787
$\Box(\Diamond\bot \to (A \land A))$	valid	25032	0.000435
$(A \lor A) \leftrightarrow (\bot \lor A)$	valid	20240	0.000357
$A \land \neg A$	invalid	7768	0.0000587
$A \lor A$	invalid	7144	0.0000495
$\Box\Box A$	invalid	31032	0.000762
$(A \lor \neg A) \land ((\neg\bot \land \neg\bot) \land A)$	invalid	12152	0.000249
$(\bot \land \bot) \lor ((A \lor A) \leftrightarrow (B \lor B))$	invalid	27864	0.000630
$\Diamond A \to \neg B$	invalid	15736	0.000236
$\Box\Box\Diamond(A \leftrightarrow A) \land \Box\Box\Diamond\Box\Diamond(A \land \neg A)$	invalid	40360	0.00123

A.2 Test set 2

$\neg(A \leftrightarrow B)$
Evaluation: invalid Memory (Bytes): 12624 Time (seconds): 0.000168

$\neg(\Box A \land \Diamond\neg A)$
Evaluation: valid Memory (Bytes): 17456 Time (seconds): 0.000343

$\neg(\Box A \leftrightarrow \Diamond\neg A)$
Evaluation: valid Memory (Bytes): 21936 Time (seconds): 0.00120

$\Box A \leftrightarrow \Diamond\neg A$
Evaluation: invalid Memory (Bytes): 17456 Time (seconds): 0.000320

$\Box(A \leftrightarrow (\Box\Box\Box\bot \to \Box\Box\bot))$
Evaluation: invalid Memory (Bytes): 189589 Time (seconds): 0.0105

$\Box(A \leftrightarrow (\Box(A \lor \Box\bot) \to \Box(A \to \Box\bot))) \to \Box(A \leftrightarrow (\Box\Box\Box\bot \to \Box\Box\bot))$
Evaluation: valid Memory (Bytes): 963852 Time (seconds): 0.292

$(((((\Box(\Box A \lor \Box\Diamond\neg A) \lor \Diamond\Box\bot) \lor \Diamond(\Box A \land \Diamond\Diamond\neg A)) \lor \Diamond(\Box\Diamond A \land \Diamond\Diamond\Box\neg A)) \lor \Diamond(\Box A \land \Box\neg A)) \lor \Diamond(\Box(\Box\neg A \lor A)$
$\land\Diamond\Diamond(\Diamond A \land \neg A))) \lor \Diamond(\Box(\Diamond\neg A \lor A) \land \Diamond\Diamond(\Box A \land \neg A))$
Evaluation: valid Memory (Bytes): 195541 Time (seconds): 0.0375

$((((((\Box(\Box A \lor \Box\Diamond\neg A) \lor \Diamond\Box\bot) \lor \Diamond(\Box A \land \Diamond\Diamond\neg A)) \land \Diamond(\Box\Diamond A \land \Diamond\Diamond\Box\neg A)) \lor \Diamond(\Box A \land \Box\neg A)) \land \Diamond(\Box(\Box\neg A \lor A)$
$\land\Diamond\Diamond(\Diamond A \land \neg A))) \lor \Diamond(\Box(\Diamond\neg A \lor A) \land \Diamond\Diamond(\Box A \land \neg A))) \lor \neg(\Box(A \leftrightarrow (\Box(A \lor \Box\bot) \to \Box(A \to \Box\bot))) \to \Box(A \leftrightarrow$
$(\Box\Box\Box\bot \to \Box\Box\bot)))$
Evaluation: - Memory (Bytes): - Time (seconds): -
