university of groningen / faculty of mathematics and natural sciences / artificial intelligence

A Tableau Prover for Non-Branching Transitive Temporal Logic with Density as Constraint

Oscar de Vries, S2713748, O.I.de.Vries@student.rug.nl,

Supervisor: prof. dr. L.C. Verbrugge

Abstract: This research aims to build a tableau-based prover for Temporal Logic. Constraints to this logic are that its models must be transitive, dense and non-branching towards the future and past. An algorithm was built in Python using a priority queue for the application of tableau-rules. Additionally the logic is proven to be sound and complete with regards to these constraints. The algorithm creates a readable human-like tableau to check the validity of input inferences. For invalid inferences, a counter-model is provided in the output. The combination of density, which can theoretically be applied infinitely many times, together with non-branching causes run-times to increase exponentially for certain inputs.

1 Introduction

In this research a program will be made which can solve inferences in temporal logic using the tableau method for this logic. Temporal logic is a form of modal logic in which the operators are with respect to moments in time. One can use this to make propositions about the future and the past. Restrictions on this logic are that it must be linear (non branching) and transitive. Also the density constraint is considered in this research. The program will be tested to see whether it is possible to function with the density restriction and if so, how this influences the performance. Next to this, the program will be compared to a program that solves inferences using a similar way. In earlier research, tableau-based algorithms were made for other types of logic, such as description logic (Baader & Sattler, 2001) and propositional dynamic logic (Abate, Goré, & Widmann, 2009).

1.1 Temporal Logic

In this Temporal Logic(TL), sometimes referred to as Tense Logic, some operators have been adjusted, such that they are easier to use than their standard modal variants. This is the case for for the operator F (used for $\langle F \rangle$), which is used to make propositions about some time in the future, and P (used for $\langle P \rangle$) for propositions about some time the past. Next to these there are also the propositions G (used for $[F]$) and H (used for $[P]$), which are used to make propositions about all times in the future and past respectively. Next to the linear and transitivity constraint, the constraint of density (δ) is considered in this research as well. Density means that between every two points in time there is another point in time, entailing that a time interval can be split up into infinitely many points (Goranko & Galton, 2015).

1.2 Tableaux for Temporal Logic

The tableau method is a formal proof procedure used for solving sentences and inferences. This method seems suitable to solve inferences in Temporal logic. In this method, a semantic tableau is built using Kripke semantics. A semantic tableau has a tree-like structure with rules for when a branch splits up into multiple branches. For inferences, the negated conclusion is put on top of the tree, together with the premises (if there are any), in order to check whether

there exists a counter-example to the inference. This is called the *initial list*. Following, eventually every branch should lead to a contradiction in order for the inference to be valid. A branch is contradicting when a proposition or atom appears on the branch together with its negation. It is then said that a branch is closed, and the full tableau is closed if every branch is closed. The tableau is said to be *complete* if there is no rule left to be applied. If at least one branch on the tableau is still open after applying all the rules, the inference is not valid. Open branches each provide a counter example for the sentence or inference (Fitting, 1999). In temporal logic, there is made use of so called worlds to denote the time of a proposition. Relations between worlds show what propositions are true in worlds prior or posterior to another. Expressions in which the 'possible' operator, such as F and P, is used in front of a proposition imply that there respectively is a future or past world in which the proposition is true, relating this the world of the expression. Expressions using the 'necessary' operator, such as G and H, in front of a proposition imply that in every future or past world respectively the proposition is true.

1.3 Tableau rules for Temporal Logic

Since TL extends the basic normal logic K, the normal K-tableau rules still apply. This means that the rules of the basic operators remain the same. These can be found in Priest (2008).

In TL, the modal operators are changed insofar as they now consider future or past worlds. In TL-tableaux this means the following: Firstly there are some rules for world-introducing formulas in which new worlds are introduced, together with a new relation. Also universal formulas are applied to existing relations:

$$\begin{array}{c} GA, i \\ irj \\ \downarrow \\ A, j \end{array}$$

the rule for GA, i should be applied to *all* j such that irj appears on the branch.

$$\begin{array}{c} FA, i \\ \downarrow \\ irj \\ A, j \end{array}$$

the rule for FA, i introduces a *new* j such that irj is introduced on the branch.

$$\begin{array}{c} HA, i \\ jri \\ \downarrow \\ A, j \end{array}$$

the rule for HA, i should be applied to *all* j such that jri appears on the branch.

$$\begin{array}{c} PA, i \\ \downarrow \\ jri \\ A, j \end{array}$$

the rule for PA, i introduces a *new* j such that jri is introduced on the branch.

Furthermore there are some rules for rewriting negations of temporal formulas:

$$\begin{array}{cccc} \neg GA, i & \neg FA, i & \neg HA, i & \neg PA, i \\ \downarrow & \downarrow & \downarrow & \downarrow \\ F\neg A, i & G\neg A, i & P\neg A, i & H\neg A, i \end{array}$$

(Priest, 2008, p50)

In this research we consider the logic to have some restrictions, which each correspond to specific tableau rules which will now be discussed.
The temporal structure is said to be *non-branching towards the future* φ *and non-branching towards the past* β:

Non-branching towards the future or forward convergent φ:

$$\forall w \forall u \forall v((wRu \wedge wRv \rightarrow (uRv \vee v = u \vee vRu))$$

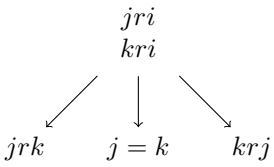
The tableau rule for φ is the following:

$$\begin{array}{ccc} & irj & \\ & irk & \\ \swarrow & \downarrow & \searrow \\ jrk & j = k & krj \end{array}$$

Non-branching towards the past or backward convergent β:

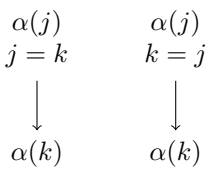
$$\forall w \forall u \forall v((uRw \wedge vRw \rightarrow (uRv \vee v = u \vee vRu))$$

The tableau rule for β is the following:



When there is an identity relation between two world (such as $j = k$ in the above examples), it means that whatever holds for a proposition α in one world (j), holds for the proposition in the other world (k).

The tableau rule for identity relation is the following:

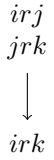


Additionally, we use the restrictions transitivity τ and density δ, which both bring a tableau rule for relations:

For a tableau to have the transitivity τ restriction, it means that the temporal structure satisfies:

$$\forall w \forall u \forall v((wRu \wedge uRv) \rightarrow wRv)$$

The tableau rule for τ is the following:

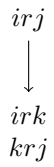


To be applied for all irj and jrk occurring on the branch. For each such irj and jrk, the rule introduces the relation irk (Priest, 2008, p38).

For a tableau to have the density δ restriction, it means that the temporal structure satisfies:

$$\forall w \forall u(wRu \rightarrow \exists v(wRv \wedge vRu))$$

The tableau rule for δ is the following:



To be applied for all irj occurring on the branch; for each such irj, the rule introduces a *new* k (Priest, 2008, p52).

2 Research question

Is it possible to make an automated tableau prover, corresponding to a sound and complete logic, for non-branching transitive Temporal Logic (TL), with density as a constraint? Also, how does this constraint influence the performance of the program? Finally, how does the program perform in comparison to a similar program in terms of user-friendliness and usefulness?

Density is a constraint that causes an infinite amount of relations between two worlds. For certain inferences, applying density can be the solution to finding a contradiction in the tableau-method, but when it does not find a contradiction, the method will keep on introducing new worlds. Since it is unwanted for a program to run infinitely, methods must be used to avoid this.
The program used for comparison is TTM by Gaintzarain, Hermo, Lucio, & Navarro (2008). This program seemed most similar the program that will be build in this research. The programs are compared on user-friendliness and usefulness. Unfortunately TTM doesn't allow for comparison of performance-time, since only an interface is provided, and not source code.

3 Soundness and Completeness

The soundness and completeness proof of propositional rules (§1.11) and K (§2.9) can be found in Priest (2008). In this proof only the characteristics added by TL are discussed. These are transitivity, non-branching, density and the new modal operators, which consider future and past.

3.1 Soundness of TL-tableaux

Proposition 1.1, Transitive For all $F =< W, R >$: ($F \models GGp \rightarrow Gp$ and $F \models HHp \rightarrow Hp$) $\Leftrightarrow$ R is Transitive

Proposition 1.2, Dense For all $F =< W, R >$: ($F \models GGp \rightarrow Gp$ and $F \models HHp \rightarrow Hp$) $\Leftrightarrow$ R is Dense
(Van Benthem, 1984)

Definition 1.1, Faithful

Let $I = < W, R, v >$ be any modal interpretation.
Let b be any branch of a 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.

Lemma 1, Soundness lemma

The propositional rules and rules for K can be found in (Priest, 2008, §1.11 and §2.9). In this proof only the additional rules for TL are considered.
Let b be any branch of a tableau, and let $I = < W, R, v >$ be any interpretation, where R is transitive, non-branching and dense. If I is faithful to b, and a TL-tableau rule is applied to it, then it produced 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. For every new tableau rule for TL it shall be shown that I is faithful to one of the resulting extended branches b'.

Suppose GA, i appears on b, and the universal rule for G is applied to obtain an extended branch b'. Since I is faithful to b, GA is true at $f(i)$. For every sentence on B of the form irj, $f(i)Rf(j)$. If irj is on b for any i and j, the branch is extended with A, j. A is true at $f(j)$ by semantics of G. Therefore I is faithful to the extension of b, b'.

Suppose HA, i appears on b, and the universal rule for H is applied to obtain an extended branch b'. Since I is faithful to b, HA is true at $f(i)$. For every sentence on B of the form jri, $f(j)Rf(i)$. If jri is on b for any j and i, the branch is extended with A, j. A is true at $f(j)$ by semantics of H. Therefore I is faithful to the extension of b, b'.

Suppose FA, i appears on b, and the world-introducing rule for F is applied to obtain an extended branch b'. Since I is faithful to b, FA is true at $f(i)$. By definition of F, there must be a new world w such that $f(i)Rw$, and A is true in w. f is then extended with $f(j) = w$. Hence $f(i)Rf(j)$ and A is true in $f(j)$. Therefore I is faithful to the extension of b, b'.

Suppose PA, i appears on b, and the world-introducing rule for P is applied to obtain an extended branch b'. Since I is faithful to b, PA is true at $f(i)$. By definition of P, there must be a new world w such that $wRf(i)$, and A is true in w. f is then extended with $f(j) = w$. Hence $f(j)Rf(i)$ and A is true in $f(j)$. Therefore I is faithful to the extension of b, b'.

Rule for τ:
Suppose that irj and jrk are on b, and we apply transitivity to get irk on extension b'. Since I is faithful to the branch, $f(i)Rf(j)$ and $f(j)Rf(k)$. By transitivity, $f(i)Rf(k)$. Therefore f shows I to be faithful to b'.

Rule for δ:
Suppose that irj is on b, and that we apply density to get irk and krj on extension b', where k is new to the branch. Since I is faithful to the branch, $f(i)Rf(j)$. By the denseness constraint, for some w, $f(i)Rw$ and $wRf(j)$. Let f' be the same as f, except that $f(k)' = w$. Since k does not occur on the branch, f' shows I to be faithful to b'.

If the rules for β and φ are applied, an extra clause for identity relations is added to the definition of faithfulness:
If $i = j$ appears on b, then $f(i)$ is $f(j)$.

Rule for φ:
Suppose that irj and irk are on b. Then $f(i)Rf(j)$ and $f(i)Rf(k)$. By the forward convergence con-

straint, $f(j)Rf(k)$ or $f(k)Rf(j)$ or $f(j) = f(k)$. So f shows at least one of the branches obtained by applying the rule to be faithful to b.

Rule for β:
Suppose that irk and jrk are on b. Then $f(i)Rf(k)$ and $f(j)Rf(k)$. By the backward convergence constraint, $f(i)Rf(j)$ or $f(j)Rf(i)$ or $f(i) = f(j)$. So f shows at least one of the branches obtained by applying the rule to be faithful to b.

Theorem 1, Soundness Theorem

If $\Sigma \vdash_{K^t_{\tau\delta\varphi\beta}} A$ *then* $\Sigma \models_{K^t_{\tau\delta\varphi\beta}} A$.
Proof:
This is a proof by contraposition.
Suppose $\Sigma \not\models_{K^t_{\tau\delta\varphi\beta}} A$. Then there is an interpretation $I = < W, R, v >$ in which R is dense, transitive and non-branching that makes A false in some world w. Let f be any function such that $f(0) = w$. This shows I to be faithful to the initial list. By repeated application of the Soundness lemma (Lemma 1), a complete branch b can be found such that I is faithful to each initial subsection of b. If b is closed, it would have to contain a contradiction in an initial subsection of b. Since I is faithful to b, this can't be the case. Hence the tableau is open. Therefore $\Sigma \nvdash_{K^t_{\tau\delta\varphi\beta}} A$.

Consequently, since
$\Sigma \not\models_{K^t_{\tau\delta\varphi\beta}} A \Rightarrow \Sigma \nvdash_{K^t_{\tau\delta\varphi\beta}} A$,
we can conclude
$\Sigma \vdash_{K^t_{\tau\delta\varphi\beta}} A \Rightarrow \Sigma \models_{K^t_{\tau\delta\varphi\beta}} A$,
therefore the logic is sound.

3.2 Completeness of TL-tableaux

Definition 2.1

Let b be an open branch of a tableau, and $I = < W, R, v >$ be the interpretation induced by b.

- $W = \{w_i : i \text{ occurs on } b\}$;
- $w_i R w_j$ 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 $v_{w_i}(p) = 0$;
  If neither p, i nor $\neg p, i$ occur on b, $v_{w_i}(p)$ can be anything one likes.

Lemma 2, Completeness Lemma

Let b be any open complete branch of a TL-tableau, and $I = < W, R, v >$ be the interpretation induced by b. Then for all (also complex) formulas A and for all i, the following hold:

- (1) If A, i is on b, then $v_{w_i}(A) = 1$;
- (2) If $\neg A, i$ is on b, then $v_{w_i}(A) = 0$.

Proof:
This will be a proof by induction on the complexity of A. The propositional rules and rules for K can be found in (Priest, 2008, §1.11 and §2.9). In this proof only the additional rules for TL are considered.

Suppose GA, i is on b. Then for all $j \in I$, such that irj is on b, A, j is on b. Hence, by construction and induction hypothesis for all w_i such that $w_i R w_j$, $v_{w_j}(A) = 1$, as required.

Suppose HA, i is on b. Then for all $j \in I$, such that jri is on b, A, j is on b. Hence, by construction and induction hypothesis for all w_i such that $w_j R w_i$, $v_{w_j}(A) = 1$, as required.

Suppose FA, i is on b. Then for one j such that irj is on b, A, j is on b. Hence, by construction and induction hypothesis for w_i, such that there is a $w_i R w_j$, $v_{w_j}(A) = 1$, as required.

Suppose PA, i is on b. Then for one j such that jri is on b, A, j is on b. Hence, by construction and induction hypothesis for w_i, such that there is a $w_j R w_i$, $v_{w_j}(A) = 1$, as required.

Rule for τ:
For $w_i, w_j, w_k \in W$ suppose that $w_i R w_j$ and $w_j R w_k$. Then irj and jrk occur on b, but then irk occurs on b (by the τ rule and the fact that b is a complete branch). Hence, $w_i R w_k$ as required.

Rule for δ:
For $w_i, w_j \in W$, suppose that $w_i R w_j$. Then irj is on b. By applying density and the fact that b is a complete branch, there is some k such that irk and krj are on b. Hence for some k, $w_i R w_k$ and $w_k R w_j$, as required.

Rule for φ:
For $w_i, w_j, w_k \in W$, suppose that $w_i R w_j$ and $w_i R w_k$ (in which i, j and k are distinct). Then irj and irk

are on b. Because forward convergence has been applied and b is a complete branch, either jrk, krj, or $j = k$ is on b; so either $w_j R w_k$ or $w_k R w_j$ or $w_j = w_k$ as required.

Rule for β:
For $w_i, w_j, w_k \in W$, suppose that $w_i R w_k$ and $w_j R w_k$ (in which i, j and k are distinct). Then irk and jrk are on b. Because backward convergence has been applied and b is a complete branch, either irj, jri, or $j = i$ is on b; so either $w_i R w_j$ or $w_j R w_i$ or $w_i = w_j$ as required.

Theorem 2, Completeness Theorem

If $\Sigma \models_{K^t_{\tau\delta\varphi\beta}} A$ *then* $\Sigma \vdash_{K^t_{\tau\delta\varphi\beta}} A$
Proof:
This is a proof by contraposition.
Suppose $\Sigma \nvdash_{K^t_{\tau\delta\varphi\beta}} A$. Let $I = < W, R, v >$ be the interpretation induced by an open branch on a complete tableau. Then, by the Completeness lemma (Lemma 2), A is false at w_0, Hence $\Sigma \not\models_{K^t_{\tau\delta\varphi\beta}} A$

Consequently, since
$\Sigma \nvdash_{K^t_{\tau\delta\varphi\beta}} A \Rightarrow \Sigma \not\models_{K^t_{\tau\delta\varphi\beta}} A$,
we can conclude
$\Sigma \models_{K^t_{\tau\delta\varphi\beta}} A \Rightarrow \Sigma \vdash_{K^t_{\tau\delta\varphi\beta}} A$,
therefore the logic is complete.

4 Method

4.1 Program design

In order to make a model out of the tableau method for temporal logic, a program was made in Python (version 3.5.2) using the steps a human being would normally also make when solving an inference with this method. This means that the program has to follow every step and expand the expressions one by one and subsequently look for contradictions. For the program to output a human-like tableau, it can't use tricks to find the correct solution without expanding every formula and check whether every branch of the tableau is closed. Tricks the program may use are rules for rewriting expressions, such as those mentioned in Section 1.3. These are rules that skip a few rewriting steps that always have to be done for such expressions, thus human solvers use them as well.

The benefit of solving the tableaux this way is that it can produce an output tableau which can be easily checked by the user.

For the program to produce a human-like tableau, it must follow the same steps a human solver would follow. This means the operations are done on the tableau with a certain order, in which there are seven different priorities. First an explanation will be given for this priority of operations, afterwards the implementation of the program will be discussed. Pseudocode of the most important functions and data structures can be found in Appendix A.

4.2 Priority of operations

In the program a priority queue is filled with operations. These operations resemble the tasks that a human solver would apply to the tableau. As long as the tableau is open and there are operations in the queue, a task is retrieved from the queue. The priority of the tasks applied to the tableau is the following:

1. Any double negations are removed from the input. Also whenever a negation is added to an already-negated expression, the program automatically removes the double negation. This is not executed as a separate step each time, therefore the tableau will never contain any double negations.

2. Non-branching expression tasks are applied, such as conjunction $\wedge$, negated disjunction $\neg\vee$ and negated implication $\neg\rightarrow$.

3. World-introducing operators are expanded, namely $F, P, \neg G$ and $\neg H$, which introduce a new relation on the branch. Subsequently the applicable universal operators $G, H\neg F$ and $\neg P$ are applied to this new relation. Also transitivity τ is checked to be applicable to this new relation. If τ forms a new relation, universals are again applied.

4. Simple branching expression are executed such as disjunction $\vee$, negated conjunction $\neg\wedge$ and implication $\rightarrow$. New branches with the disjuncts are added to all leaves of the branch on which the expression is found.

5. Complex branching expressions are executed, such as bi-implication $\leftrightarrow$ and negated bi-implication $\neg \leftrightarrow$. They are complex insofar as they add two expressions to each branch, whereas a simple branching expression only adds one. Also the branching happens at all leaves of the branch of the corresponding expression.

6. The linearity restriction is applied, which is forward convergence φ and backward convergence β. For every new relation (irj), it is checked whether it has either the same i or the same j as another relation on the branch. A relation pair causes three new branches at every leaf node.

7. The density restriction δ is applied, adding two new relations to the branch.

These operations make up the majority of the program. Now the implementation of the program will be discussed.

4.3 Procedure

4.3.1 Input parsing

An input is given in the terminal in the form of p_1, p_2,...,p_n -> c in which p_1, p_2,...,p_n are the premises and c is the conclusion, both in infix notation. Solely a conclusion is also a valid input. Any double negations in the string are removed at this point. The program parses this by splitting the conclusion from the premises by splitting string at the '->' sign. The premises are separated by splitting the remaining string at every comma ','. Each premise together with the negated conclusion is parsed into an expression tree with both unary and binary nodes.

The input is expected to have parentheses around every binary operator and its operands, and not around unary operators and its operand. This way the parser knows when a node in the tree should be unary and when it should be binary. The expression trees are put in a data structure together with an integer for the world of the expression. The input is always put in world 0. A Tableau-node is created, and all expressions (tree + world) are added to the expression-list of this node. This node will be the root of the tableau, and will be used to access the tableau in the rest of the program.

The use of an expression-list in a node rather than a node for each expression seemed convenient, since this makes it easier to traverse the data structure of the tableau. For example, when multiple disjunctive expressions are expanded in a node, only a few visits are necessary to reach leaf-nodes. A single node per expression would mean that every expression is visited each time such an expression is expanded. Also an expression list is clearer, as it resembles the human-made tableau. A node per expression gives a certain order to the expressions, which is not necessarily present in a human-made tableau.

4.3.2 Task checker and contradiction check

The root is then checked for tasks by going through the expression-list and adding a task for each expression that isn't already checked for a task. It adds the task by determining the main operator of the tree and checking the priority of this operator. A queue-item is created, consisting of the Tableau-node, the index of the expression on the expression-list of the Tableau-node and the priority. This queue-item is put in a priority queue, based on the priority of the main operator.

The expression-list on the root is checked for contradictions. It does so by checking whether an expression-tree is equal to another expression-tree on the branch, with one of them being negated. If this is the case and also the world is equal, the node closed. If this is the case for the root no tasks need to be executed and the tableau closes. From this point onward, the program goes into a while-loop as long as there are items in the queue. In each iteration the following steps are executed:

1. Check whether the tableau is closed, by checking if all leaf-nodes are closed. If it's closed, the program breaks out of this loop.

2. Pop an item from the priority queue.

3. Check for a pattern indicating infinite open branches. Break from loop if this is the case.

4. Select the right task to apply, according to the priority of the queue item.

5. Apply the task on the queue item (tasks are discussed more in depth later on).

6. Check whether the added formulas from the task cause contradictions on the branches they are added to. Close the leaf-nodes accordingly.

Since linearity has a higher priority than density, sometimes a contradiction is found in nodes that already have children. In these cases first linearity was applied, but later on a contradiction is found in a non-leaf node, since density can be applied on such nodes. In these cases every leaf-node of the corresponding node is closed. Hence the full tableau can still be closed by looking at all the leaf nodes.

4.3.3 Expression tasks

Expression tasks are handled as follows: in the task-function, the expression tree is derived from the Tableau-node and expression index given in the arguments. The tree is then analysed for the main operator; according to this, the children (left/right or middle if main operator is unary) are determined. These can be negated or non-negated, depending on the operator. The children are then added to either the same Tableau-node for non-branching tasks, or to all the open leaves of the node for branching tasks (leaves are the ends of a branch; they are nodes that aren't split, thus do not have child nodes yet). All adjusted or newly added expressions are checked for contradictions on their branch. This has the advantage that you can close this node immediately. New expressions are then checked for new tasks, which are added to the queue.

4.3.4 World-introducing tasks

World-introducing tasks (F and P) introduce a new relation on the branch, and add the expression to the new world. Subsequently all the applicable universal expressions are applied to the newly introduced relation. The new relation is checked with other relations on the branch for transitivity (τ). If transitivity can be applied to the new relation, it is done at this point. Applicable universal expressions are applied to the transitive relation. The expressions are checked for contradictions once more. New expressions and the new relation are checked for tasks to put on the queue.

4.3.5 Relation tasks

Relation tasks (δ, φ and β) work as follows. A Tableau-node and an index for the relation location are given for density (δ). For every relation on the branch, two new relations are added to the branch in which a new world is introduced between the two worlds of the old relation.

For linearity tasks, a relation pair is given from the queue, in which either φ or β is applicable. If so, the corresponding leaves of tableau split into three branches, each with a new relation. One has a identity relation in which the non-common part of the two relations is equal ($i = j$) and in the other two branches either one of the non-common parts of the two relations get a relation to one another (irj) or (jri). For an identity relation, all expressions which have either of the worlds will also be added to the branch with the other world. For all other new relations in both density and linearity, universals are again applied. New expressions and new relations add tasks to the queue once again.

4.3.6 Validity of the inference

Either the queue completely finishes, meaning all tasks are handled, or there is an early break. A break happens when either the tableau is closed, or when density-tasks seem to go on infinitely. When a tableau is closed, the conclusion can be made that the inference is valid. Otherwise it isn't valid and a counter-model must be provided. The tableau traverses the branch by means of Depth-First-Search, going to the most left leaf first, then proceeding to the most right leaf of the tableau. The first open branch it encounters gets checked for literals, and the corresponding relations that form the relations between the literals. From this a counter model is generated of the form $< W, R, v >$. This suffices as a counter-model.

4.3.7 Infinite applications of density

Since applying the density rule can theoretically be done infinitely many times, a check must be implemented to stop this. In the program this happens when a density task is executed consecutively for a certain number of times. Since linearity is a higher priority than density, and density tasks will in most cases add a linearity task to the queue, a pattern will occur in which each time density is executed,

after which it is followed by a couple of linearity tasks. Therefore the check ignores linearity tasks when counting consecutive density tasks.

The exact pattern is that the first time density is applied, linearity applies two times. Following, when density is applied again, linearity applies 3 times. After this, when density is applied again, linearity applies 4 times. Finally the pattern seems to alternate between one application of density, followed by either 3 or 4 applications of linearity. Since linearity splits every leaf into three new branches, this process becomes very time consuming when applied many times. Together with the idea that applying linearity multiple times makes it highly unlikely to find a definite answer in the sense that a branch closes, this results in the choice that after 4 consecutive applications of linearity, the program no longer applies linearity, given that it has run for at least a set amount of time. Consequently, the program finds an unbounded open branch due to density a lot quicker in such cases.

The number of times density is allowed to be executed consecutively can be manually set, but this would make it arbitrary. Thus the number is determined by the input sentence as follows: In this sentence all the arguments are checked for universal operators (G and H). For every argument, the number of universal operators is counted, when at least one occurs in the argument. The program keeps track of the highest and the lowest number of occurrences of G and H in an argument. After this is done for each argument, the highest difference between all the arguments is computed.

For example, in the input sentence ($GGGp \rightarrow Gp$) the difference is 2, and in the sentences $HHq, HHHHHHHr \rightarrow Hr$ the difference is 6. Only the operators are considered for this method, as looking at the following propositions would be using a non human-like trick. Therefore also the sentence $HHHHHHHp \rightarrow Hq$ would result in a difference of 6.

This method is chosen because the density constraint can in some cases be the crucial step to finding two contradicting formulas on a branch. An example of a sentence solved by applying density can be found in Figure 1.
In the first example $GGGp \rightarrow Gp$, density adds worlds in between two worlds of a relation, allowing the inference to be valid. Therefore, density should be allowed to occur at least the number of times of the maximal difference of universal operators, as discussed earlier. In the program it was chosen that density can always occur at least 4 times consecutively. Whenever the difference turns out to be higher than 4, that number will become the number of allowed consecutive occurrences of density.

Semantic tableau:

$$GGGp,\ 0$$
$$\neg GGp,\ 0$$
$$\neg Gp,\ 1$$
$$GGp,\ 1$$
$$\neg p,\ 2$$
$$Gp,\ 2$$
$$GGp,\ 2$$
$$GGp,\ 3$$
$$Gp,\ 1$$
$$0r1$$
$$1r2$$
$$0r2\ \tau$$
$$0r3\ \delta$$
$$3r1\ \delta$$
$$|$$
$$\times$$

Tableau closed, inference is valid

Figure 1: Output for the input sentence $GGGp \vdash_{K^t_{\tau\delta\varphi\beta}} GGp$

4.3.8 Output

The number of steps together with the execution time of the program is printed on the terminal. The program creates a LaTeX-file from the full tableau. In this file the tableau is visualised as if it were human-made. On the tableau every relation shows whether it is introduced by one of the restrictions. If infinite applications of density is the reason for the open tableau, an infinite sign is placed on that branch. Closed branches are followed up by a cross. A counter model is provided on the output file whenever an inference is not valid. The counter model is an interpretation ($I =< W, R, v >$) of an open branch, with:

- W: Worlds of literals on the branch
- R: Relations between literals on the branch and
- v: Valuation of literals.

Input:
$Ga, (b \vee c), (d \to e), Fp \vdash_{K^t_{\tau\delta\varphi\beta}} Hq.$

Tableau open, inference is not valid

Counter-Model, a dense and transitive model $< W, R, v >$ read off from open branch:

W:
$w_0, \quad w_1, \quad w_2, \quad w_3, \quad \ldots \quad , \quad \infty$
R:
$w_0 R w_1\,, \quad w_2 R w_0\,, \quad w_2 R w_1\,(\tau), \quad w_0 R w_3\,(\delta),$
$w_3 R w_1\,(\delta), \quad w_2 R w_4\,(\delta), \quad w_4 R w_0\,(\delta)$
v:
$v_{w_2}(q) = 0, \quad v_{w_1}(a) = 1, \quad v_{w_1}(p) = 1,$
$v_{w_0}(b) = 1, \quad v_{w_3}(a) = 1, \quad v_{w_0}(d) = 0$
R is the dense and transitive closure of $[w_2, ..., w_1]$ as in Q
$w_2 \to .... \to w_4 \to .... \to w_0 \to .... \to w_3 \to .... \to w_1$

Figure 2: Example of a counter-model, the according Tableau is too big to print in LaTeX

An example of a counter-model can be found in Figure 2. The tableau of the inference in Figure 2 is left out, as it is too large to be printed by LaTeX.
In the case that infinite applications of density caused the main loop to break, the output is supplemented by the worlds of the relation between which the density holds. Theoretically, every relation is dense, because of this only the first relation in which density is applied is provided. This file is then automatically compiled and the output PDF-file is opened to immediately show the tableau for the given input. Examples of output, regarding the different constraints can be found in Appendix B, Figures B.1-B.4.

5 Results

5.1 Removing duplicate expressions

Since applying density causes the application of linearity, the run time of certain input inferences increases a lot. An explanation for this is that inferences including universal operators will cause the patterns in which linearity is applied. Because of this, the branch on which the identity relation is introduced will add many expressions that are already on the branch, especially when linearity is applied multiple times. Consequently the number of expression with universal operators increases exponentially. Therefore a check was implemented in which a branch was checked before putting an expression on it. If the expression was already on the branch, it is not added again.
To see how this check influences the number of steps and the amount of run-time needed to find a solution, a set of inferences was tested both with the check and without the check. The inferences consisted only of valid inferences, as invalid inferences in which a universal operator appears automatically lead to infinite applications of density, which does not produce a useful result. Also no complex expressions are used, but only literals. This allows to see exactly how the number of universal operators influences the run time.
In Figure 3 the number of steps needed to reach a conclusion are plotted against the number of universal operators in the inference. Here it can be seen that the number of steps needed increases exponentially as the number of universal operators increases. With at least 3 universal operators in the inference, the number of steps needed is larger when duplicate expressions are not removed.

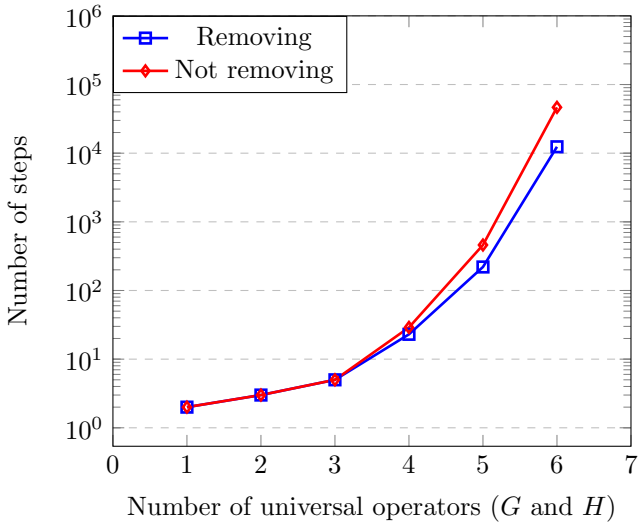


Figure 3: Number of steps needed to find a solution for valid inferences, with or without checking and removing duplicate expressions.

In Figure 4 the amount of time (in ms) needed to reach a conclusion is plotted against the number of universal operators in the inference. For each inference except the one with 6 operators, the program was run 100 times and the average run-time was taken. This is done since for small inferences the run-time can differ per run, therefore averaging would give a more reliable answer. For the inference with 5 operators, 100 runs took about 10 seconds in

the case of not removing duplicate expressions. Since running the program for 100 times with 6 operators would take too much time, it was decided to run this inference 10 times, which took about 13 minutes to compute.
Another thing that stands out in Figure 4 is that checking for duplicate expressions causes the program to run longer for 2 universal operators. The run-time of solving the problem here is around the same length of the run-time of checking for duplicate expressions. Thus checking for duplicate formulas is less efficient for inferences of this size.

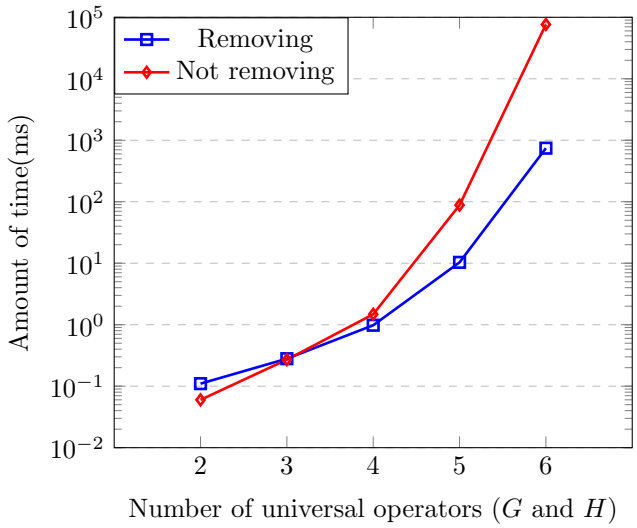


Figure 4: Amount of time needed to find a solution for valid inferences, with or without checking and removing duplicate expressions.

6 Comparison to TTM

Our program (from hereon TLTPδ, for Temporal Logic Tableau Prover, dense) is compared to the program TTM: A Tableau-based Theorem Prover for Temporal Logic PLTL, by Gaintzarain et al. (2008). For this program, an interface is provided in which inferences in propositional temporal logic can be tested for satisfiability. TTM is slightly different insofar as it only considers the future, thus only uses the G and F operators. The comparison is made on the following points: User-friendliness and usefulness.

For user-friendliness, TTM has the advantage that a user interface has been made, in which you can load a set of inferences or manually add them to create a set. In TLTPδ only one inference can be tested at a time. This inference is the input given to the terminal. The interface does crash occasionally, it is then not clear why it does so.
Both TTM and TLTPδ have the disadvantage that input is dependent on parentheses. For larger inferences this means the user has to keep track of having the right number of parentheses at the correct positions, else the program isn't able to parse the inference.
TTM has a description of the syntax in the user interface, which gives all possible operators to use in the formulas. However, this does not describe how parentheses should be placed. An improvement for both programs would be a function that manages parentheses and corrects whenever the number of parentheses is incorrect.
In terms of usefulness TLTPδ has a clear answer about satisfiability of the input inference. TTM does not describe whether an inference is valid, but gives the number of open and closed branches. It is then not concluded whether or not this means the inference was valid. TTM could improve this by drawing a conclusion from the answers, or explaining what can be said about validity from the answers. TLTPδ provides the user with the conclusion in the terminal. Next to this TLTPδ provides a counter example in the terminal whenever an inference isn't valid. In this terminal output also the run-time and the number of expressions expanded is given, this allows for easy experimentation.
On top of this, TLTPδ makes the PDF-output file in which the tableau is printed, together with (possibly) a counter-example read from an open branch, in the form of an interpretation ($I =< W, R, v >$). Advantage of this is that the user can easily see the reasoning of the program in the tableau-output and check whether the given counter-example is actually read from the tableau. TTM is tableau-based, but does not give the opportunity to check an actual tableau for a given inference. An improvement here would be to visualise the tableau and give the user the opportunity to check a representation of the reasoning.
TTM only works for inferences about the future, whereas TLTPδ also includes inferences about the past. TTM does have two additional operators, namely X (for next) and U (for until). This gives TTM the advantage that more inferences about the future are possible than in TLTPδ.

7 Discussion

A program was made, which is able to proof the validity of TL-inferences by means of the tableau method, constructing and visualising the tableau, and possibly a counter-model, for the user to examine. The rules and constraints considered in TL were proven to be sound and complete. Constraints were that the logic had to be transitive, non-branching and dense, the latter of which may theoretically need infinitely many applications of the corresponding tableau-rule. It was researched how this influences the performance of the program. It turned out certain patterns occurred, due to the priority of operations, signifying an infinite application of operations.

The same pattern occurred for certain valid inputs containing universal operators, causing long run-times due to an inefficient priority of operations. A check was implemented in which a branch of the tableau was inspected before an expression was put on it to see whether it already existed on the branch. This check notably reduced the number of steps needed for finding answers to these inputs and consequently the run-time was reduced as well.

The difference in performance can be explained by the priority of operations, as non-branching (linearity) is applied before density within the specific pattern. For certain inputs that require applications of density to solve them, linearity is automatically applied. This causes the leaf-nodes to split into three branches each time, resulting in an exponential increase in run-time. Also due to the identity relation occurring on one branch, duplicates of the same expression were added on a branch. Therefore removing these duplicates improved the performance.

The program worked well for relatively small inferences, as these resulted in a readable tableau as output. In this output the steps a human solver would also apply can be easily followed. Bigger inferences, especially including universal operators, resulted in longer run-times and bigger tableaux. These were harder to read or could sometimes not even be printed by LaTeX.

7.1 Problems

Since this method is based on the order of operations a human solver would use, it does not use the ideal priority for each inference. This can result in inefficient solving, usually by many applications of linearity. In cases in which the inference should be solved by applying density, the program applies linearity first, as of its priority. The leaves of the tableau can split multiple times into three new branches, while the contradiction is eventually found in the first node. This is the consequent of solving the tableau in a human-like way, as this will always result in a trade-off between efficiency and resemblance to human solving. An efficient solver for inferences would probably use more non-human analysis during the solving, to execute the most efficient step at each iteration. This could result in more efficient solving, but at the cost of making the output less readable.

7.2 Possible improvements

A possible improvement to the program would be a function which manages the parentheses of the input. Since the parser is now dependent on parentheses around binary operators, this can make it difficult to keep track of the parentheses. A function that either gives feedback to the user, or is able to fix an input inference would be an improvement.

Another improvement would be a function that keeps track of the size of the output tableau, such that it does not exceed the limits of a LaTeXfile. Consequently, when a large tableau is computed it could still be printed. Instead of printing each expression, it could indicate oversized branches another way, and only print either the open branch read for the counter model. In the case of a closed tableau which is too large to print wholly, it could print only the expressions causing the contradiction for each branch.

References

Abate, P., Goré, R., & Widmann, F. (2009). An on-the-fly tableau-based decision procedure for PDL-satisfiability. *Electronic Notes in Theoretical Computer Science*, *231*, 191–209. doi: 10.1016/j.entcs.2009.02.036

Baader, F., & Sattler, U. (2001). An overview of tableau algorithms for description logics. *Studia Logica*, *69*(1), 5–40. doi: 10.1023/A:1013882326814

Fitting, M. (1999). *Handbook of Tableau Methods*. doi: 10.1007/978-94-017-1754-0

Gaintzarain, J., Hermo, M., Lucio, P., & Navarro, M. (2008). Systematic semantic tableaux for PLTL. *Electronic Notes in Theoretical Computer Science*, *206*(C), 59–73. doi: 10.1016/j.entcs.2008.03.075

Goranko, V., & Galton, A. (2015). Temporal logic. https://plato.stanford.edu/archives/win2015/entries/logic-temporal/.

Priest, G. (2008). *An Introduction to Non-Classical Logic: From If to Is* (2nd ed.). Cambridge University Press. doi: 10.1017/CBO9780511801174

Van Benthem, J. (1984). Correspondence theory. In D. Gabbay & F. Guenthner (Eds.), *Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic* (pp. 167–247). Dordrecht: Springer Netherlands. doi: 10.1007/978-94-009-6259-0_4

Appendix A

A1. Input for the Parser

Operator	Input
$\neg$	$\sim$
$\wedge$	&
$\vee$	|
$\rightarrow$	>
$\leftrightarrow$	=
[P]	H
$\langle P \rangle$	P
[F]	G
$\langle F \rangle$	F
$\bot$	#
$\vdash$	->

Atoms are all non-capital letters of the alphabet

A2. Priority of operations

Priority 1

- $\neg\neg$

Priority 2

- $\wedge, \quad \neg\vee, \quad \neg\rightarrow$

Priority 3

- $F, P, \neg G$ and $\neg H$ (world introducing rules, for new relations, it is checked whether τ is applicable)
- $G, H \neg F$ and $\neg P$ are applied whenever a new relation is placed on the branch that applies to that specific proposition (universal rules).

Priority 4

- $\vee, \quad \neg\wedge, \quad \rightarrow$

Priority 5

- $\leftrightarrow, \quad \neg\leftrightarrow$

Priority 6

- $\varphi, \quad \beta$

Priority 7

- δ

A priority queue is based on this hierarchy of operations

A3. Abstract data types:

TableauNode	
List expressionList, relationList	▷ List of all expressions on the node and a list of all relations on the node
TableauNode leftChild, rightChild, middleChild, parent	▷ middleChild is used when linearity is applied, then the TableauNode splits into three branches
Boolean closed	▷ TableauNode is closed whenever a contradiction has appeared

Expression	
String expression	
Int world	
ExpTree expressionTree	▷ Binary/Unary tree
Boolean closed, taskAdded	

Relation	
Int world1, world2	▷ relation from world1 to world2
Boolean equal	▷ true when an identity relation $(i = j)$ is added from linearity

ExpTree	
Char value	▷ value can be a proposition or an operator
ExpTree left, right, parent, child	▷ child is used when the operator is unary
Boolean unary	

QueueItem	
	▷ Priority queue item
Int priority	▷ Based on the priority of the main operator
TableauNode node	
ExpLocation expression	▷ pointer to the correct index of expressionList in TableauNode

A4. Functions

Algorithm 1 Parse Input ▷ Constructs Expression Trees for the premises and negated conclusion in the input and puts them in the root of the Tableau

```
1: procedure ParseInput(input)
2:     root ← TableauNode
3:     ConstructTree(Premises, negatedConclusion)
4:     root.expressionList ← premises, negatedConclusion
5:     return root
6: end procedure
```

Algorithm 2 Main

```
1: procedure Main
2:     root ← ParseInput(input)
3:     addTasksFromNode(root)                          ▷ adds tasks to priority queue
4:     while queue not empty do
5:         checkContradictions()
6:         if checkTableauClosed then
7:             break
8:         end if
9:         item ← pop(priorityQueue)
10:        applyTask(item.priority, item.expression)
11:    end while
12:    checkContradictions()                           ▷ final queue item can cause contradictions
13:    checkTableauClosed()
14:    showTableau()
15:    if Tableau not closed then
16:        findOpenBranch(Tableau)                     ▷ Using DFS find first non-closed leaf node
17:        findCounterModel(OpenBranch)
18:    end if
19: end procedure
```

Algorithm 3 Add Tasks From Node ▷ Applied when root is created and whenever a new expression is added on the Tableau

```
1: procedure addTasksFromNode(Node)
2:     for Expressions on node do
3:         if Not Task added then
4:             queue ← addTaskToQueue(Expression, priority)
5:         end if
6:     end for
7: end procedure
```

Algorithm 4 Check Contradictions ▷ Checks whether a contradiction appears on a leaf node, and closes leaf if this is the case

```
1: procedure checkContradictions(Node)
2:     for Expressions on branch do
3:         if Expression and Negated expression on branch then ▷ World must equal as well
4:             closeNode()
5:         end if
6:     end for
7: end procedure
```

Algorithm 5 Check Tableau Closed ▷ Checks whether all branches are closed, if one isn't closed, the Tableau is not closed

```
1: procedure checkTableauClosed(Tableau)
2:     for Leaf-nodes on tableau do
3:         if Leaf not closed then
4:             return False
5:         end if
6:     end for
7:     return True
8: end procedure
```

Algorithm 6 Non Modal Task ▷ Routine for tasks that are not modal and do not involve relations

```
1: procedure NonModalTask(TableauNode, ExpLocation)
2:     Expression ← TableaNode.Expressionlist[ExpLocation] ▷ Expression Tree
3:     NewExpressions ← Tree.leftChild, Tree.rightChild ▷ Main operator is on top of the tree,
       leftChild and rightChild are new expressions
4:     if BranchingExpression then ▷ Disjunctive tasks, add disjuncts to leaves
5:         for Leaves of TableauNode do
6:             Leaf.addLeftChild(expressionLeft) ▷ expressions are only added if node isn't closed
7:             Leaf.addRightChild(expressionRight)
8:             checkContradictions()
9:             addTasksFromNode(leaf.rightChild)
10:            addTasksFromNode(leaf.leftChild)
11:        end for
12:    else ▷ Conjunctive Tasks, add conjuncts to same node
13:        TableauNode.addExpression(expressionLeft, expressionRight)
14:        checkContradictions()
15:        addTasksFromNode(TableauNode)
16:    end if
17: end procedure
```

Algorithm 7 World Introducing Task ▷ Introduces new world (for operators such as $F, P, \neg G$ and $\neg H$)

procedure ModalTask(TableauNode, ExpLocation)
  $Expression \leftarrow TableaNode.Expressionlist[ExpLocation]$ ▷ Expression Tree
  $newRelation \leftarrow Relation(Expression.world, newWorld)$
  $TableauNode.relationList \leftarrow newRelation$ ▷ newWorld is a new int
  $TableauNode.addExpression(ExpressionChild, newWorld)$ ▷ ExpressionChild can be negated
  $checkContradictions()$
  $applyUniversals(newRelation)$
  $checkIfConstraintApplicable(newRelation)$ ▷ Check if transitivity is applicable, and adds task for density/linearity
  $addTasksFromNode(TableauNode)$
end procedure

Algorithm 8 Apply Universals ▷ Applies all universals applicable to new relation (for operators such as $G, H\neg F$ and $\neg P$)

procedure ApplyUniversals(Relation)
  for Universals on non-closed branch **do**
    if Relation.world1 equal to Universal.world **then**
      $TableauNode.addExpression(UniversalChild, newWorld)$ ▷ UniversalChild can be negated
      $checkContradictions()$
      $addTasksFromNode(TableauNode)$
    end if
  end for
end procedure

Algorithm 9 Apply Transitivity ▷ $\forall w \forall u \forall v((wRu \wedge uRv) \rightarrow wRv)$ Create new transitive relation, check this relation for universals applicable and contradictions

procedure ApplyTransitivity(TableauNode, inputRelation)
  for Relations on branch **do**
    if inputRelation.world1 equal to Relation.world2 **then**
      $newRelation \leftarrow Relation(Relation.world1, inputRelation.world2)$
      $TableauNode.relationList \leftarrow newRelation$
      $applyUniversals(newRelation)$
      $checkContradictions(Tableaunode)$
      $checkIfConstraintApplicable(newRelation)$ ▷ Adds task for density/linearity
    end if
  end for
end procedure

Algorithm 10 Apply density ▷ $\forall w \forall u(wRu \rightarrow \exists v(wRv \wedge vRu))$ Create new world in between the worlds of the input relation, then check new relations for universals applicable and contradictions)

procedure ApplyTransitivity(TableauNode, Relation)
    $newRelation1 \leftarrow Relation(Relation.world1, newWorld)$
    $newRelation2 \leftarrow Relation(newWorld, Relation.world2)$
    $TableauNode.relationList \leftarrow newRelation1$
    $TableauNode.relationList \leftarrow newRelation2$
    $applyUniversals(newRelation1)$
    $applyUniversals(newRelation2)$
    $checkIfConstraintApplicable(newRelation1)$ ▷ Adds task for density/linearity
    $checkIfConstraintApplicable(newRelation2)$
    $checkContradictions(Tableaunode)$
end procedure

Algorithm 11 Apply linearity ▷ For two relations that either have the same first or second world, apply backward/forward convergence to leaf nodes)

procedure ApplyLinearity(TableauNode, Relation1, relation2)
    New relations formed from relation1 and relation2, depending on φ or β
    for Leaf Nodes **do**
        $AddThreeChildren(leaf)$ ▷ left,right and middle child
        for Children **do**
            $AddRelation$ ▷ left gets $i = j$, middle gets irj, right gets jri
        end for
    end for
end procedure

Appendix B

B1. Example output

Input:

$G(a \to b), Fa \vdash_{K^t_{\tau\delta\varphi\beta}} Fb.$

Semantic tableau:

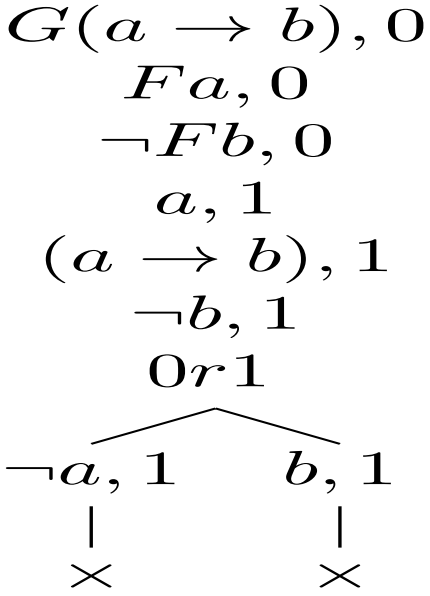


Tableau closed, inference is valid

Figure B.1: Example Temporalized Modus Ponens

Input:

$GGa \vdash_{K^t_{\tau\delta\varphi\beta}} Ga.$

Semantic tableau:

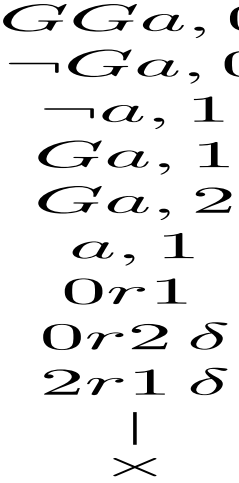


Tableau closed, inference is valid

Figure B.2: Check for Density

Check for Transitivity:

Input:

$Ha \vdash_{K^t_{\tau\delta\varphi\beta}} HHa.$

Semantic tableau:

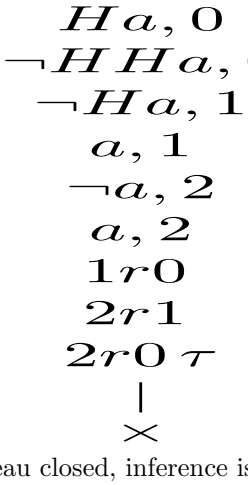


Tableau closed, inference is valid

Figure B.3: Check for Transitivity

Check for linearity:

Input:

$(Fp \wedge Fq) \vdash_{K^t_{\tau\delta\varphi\beta}} ((F(p \wedge q) \vee F(Fp \wedge q)) \vee F(p \wedge Fq))$.

Semantic tableau:

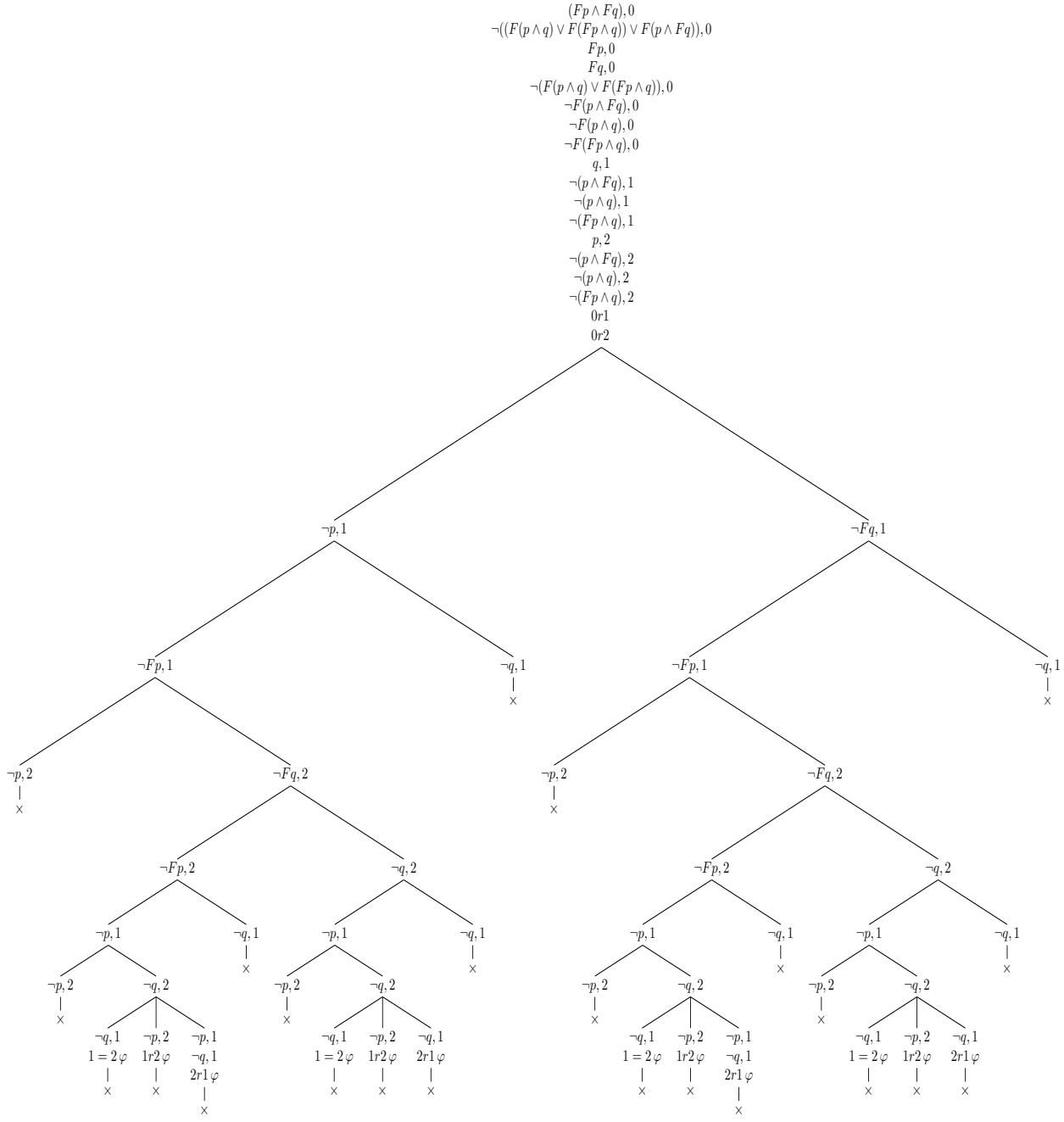


Tableau closed, inference is valid

Figure B.4: Check for Linearity