6CCS3PRJ
Logic of Gossip

Final Project Report

Author: Dariana Dorin

Supervisor: Christopher Hampson

Student ID: 20037673

April 6, 2023

Abstract

This project explores the logical foundations of gossip through the application of modal logic and the tableau algorithm. The aim is to establish a formal framework for reasoning about gossip and its validity. The software solution implemented allows for the automated generation of tableaux and visualization of counter models for invalid formulas. This project also provides a comprehensive overview of the theories of modal logic therefore is serves as a great starting point for more advanced topics.

Originality Avowal

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Christopher Hampson, for his invaluable guidance and support throughout the entire project.

Contents

Chapter 1

Introduction

The exchange of rumours and gossip among individuals is a frequent occurrence in human societies. Examining gossip can provide insight into various social phenomena, such as the dissemination of information and the establishment of social connections. Despite its significance, the principles governing gossip have been given little attention in scholarly literature. Analysing the logical foundations of gossip is not only relevant to philosophy and social science but also to computer science.

Modal logic is a branch of logic that extends classical propositional and predicate logic by adding operators that express modalities, such as possibility, necessity, and belief. These operators enable us to reason about how propositions can be true or false in different possible worlds. In this paper, we will focus on using modal logic to clarify the concept of gossip and absolute truth via validity.

1.1 Motivation

For this project, we will demonstrate a Python-based software implementation of the validity of modal logic formulas algorithm. This implementation enables the automated generation of tableaux and visualization of counter models for different gossip scenarios, enabling the assessment of logical properties of gossip statements. The software will have practical applications in evaluating the logical validity of gossip statements, as well as analysing the propagation of information within social networks.

We aim to present a comprehensive overview of the tableau algorithm and its practical application in studying gossip. This algorithm will serve as a valuable tool for exploring the logical foundations of gossip and generating counter-models in the event of invalidity. Therefore our objective is to establish a formal framework for reasoning about gossip using modal logic. With the advent of the internet and social media, the proliferation of information and the formation of societies have become increasingly relevant in the field of computer science. Understanding the underlying logic of gossip and how information spreads in different realities has practical implications in areas such as network security, information retrieval, and natural language processing.

The field of artificial intelligence heavily relies on the ability to model and comprehend the propagation of information in social networks, as this knowledge is necessary for creating intelligent agents that can efficiently navigate and make use of these networks. Moreover, the modal logic theories we are about to present have broader applications in natural language processing. Specifically, these tools can facilitate the analysis of the certainty and possibility of statements, thereby contributing to the development of more advanced natural language processing techniques.

Chapter 2

Background

The study of logic is the study of reasoning and argumentation. Logic can be divided into two main branches: propositional logic and predicate logic. Propositional logic deals with statements that are either true or false, while predicate logic deals with statements that contain variables and quantifiers. The logic of gossip is a form of modal logic, which is an extension of predicate logic that deals with statements about possible worlds and the accessibility relations between them. Modal logic is particularly well-suited for modelling gossip because it allows us to reason about the different ways in which information can be spread and the different levels of certainty that we can have about the truth of a statement. In modal logic, a statement is evaluated not just in terms of its truth value but also in terms of the accessibility of the worlds in which it is true.

Naming and Necessity[8], a major work by Saul Kripke, gave a revolutionary perspective on modal logic. Kripke suggested that modal statements, such as those involving possibility and necessity, have a metaphysical basis as well as an epistemic or linguistic grounding. He stated that the meaning of proper names, and thus modal statements, cannot be reduced to descriptive or referential semantics. Kripke's modal logic is based on the concept of potential worlds, with each world representing a unique way the universe could be. For example, a statement like ”It is possible that X is cheating on Y” can be evaluated as true in some possible worlds and false in others, depending on whether X is unfaithful there, whereas the statement ”water is H2O” is a necessary truth, as it is true in all possible worlds.

2.1 Logic types

When it comes to logical modelling, there are three key components to consider: syntax, semantics, and the connection between them. Syntax involves using precise formal language to express statements, while semantics involves interpreting those expressions. Once we have these components set up correctly, we can evaluate the validity, satisfiability, logical consequence, and consistency of any logical problem.

2.1.1 Propositional logic

Propositional logic, also known as Boolean logic, is a branch of mathematical logic that deals with propositions, which are statements that can be evaluated as true or false. In propositional logic, propositions are combined using logical connectives, including negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and bi-implication ($\leftrightarrow$).

These logical connectives allow us to form more complex statements, known as well-formed formulas (WFFs), by combining simple propositions. The truth values of the propositions used to form this, along with the way they are combined using logical connectives, determine the truth value of the WFF as a whole. In this way, propositional logic provides a powerful tool for modelling and reasoning about relationships between statements, and it forms the foundation of many other branches of logic, including modal logic.

Let's consider the well-formed formula "$p \wedge (q \rightarrow \neg r)$". To evaluate the truth value of this WFF, we can use truth tables. A truth table is a table that lists all possible combinations of truth values for the propositions that make up a formula, along with the corresponding truth value of the formula. In this case, we have three propositions: p, q, and r. Each of these propositions can either be true or false, which means there are $2^3 = 8$ possible combinations of truth values for these propositions.

p	q	r	q $\rightarrow \neg$r	p $\wedge$(q $\rightarrow \neg$r)
T	T	T	F	F
T	T	F	T	T
T	F	T	F	F
T	F	F	T	T
F	T	T	F	F
F	T	F	T	F
F	F	T	F	F
F	F	F	T	F

Conjunction and disjunction work out as the English words “and” and “or”. An example of using ”implies” in propositional logic would be the statement ”If Ana has money, then Ana was not fired from her job.” This statement can be represented as ”Ana has money $\rightarrow \neg$(Ana was fired from her job)”. To determine the truth value of this statement, we need to consider the truth values of the two propositions. If ”Ana has money” is true and ”Ana was fired from her job” is false, then the whole statement is true. If ”Ana has money” is false, the statement is true regardless of the truth value of ”Ana was fired from her job”. If ”Ana has money” is true and ”Ana was fired from her job” is true, then the whole statement is false.

A formula is considered satisfiable if there is at least one assignment of truth values to its propositional variables that makes the formula true. This means that the formula is true under at least one interpretation or model. Conversely, if there is no such assignment that makes the formula true, then the formula is unsatisfiable. A formula is considered valid if it is true under all possible assignments of truth values to its propositional variables. This means that the formula is true under every possible interpretation or model. Conversely, if there is at least one assignment of truth values that makes the formula false, then the formula is invalid. Therefore a true formula is a formula that is always true, regardless of the interpretation or model used. In other words, a true formula is a formula that is valid. Similarly, a false formula is a formula that is always false so that is unsatisfiable.

2.1.2 Predicate logic

Predicate logic is a subfield of logic that deals with quantifiers and variable-containing propositions. Predicate logic allows us to make claims about variables and their attributes, unlike propositional logic, which exclusively deals with statements that are either true or untrue. In predicate logic, predicates are used to specify characteristics or connections among variables. A predicate is a function that accepts inputs in the form of variables and outputs a truth value. If a person x is unfaithful to his wife, for instance, the predicate ”IsUnfaithful(x)” is true for that individual x.

Predicate logic uses quantifiers in addition to predicates to define the generality of statements. The two most common quantifiers are the universal quantifier ”$\forall$” (reads as ”for all”) and the existential quantifier ”$\exists$” (reads as ”there exists”). For example, the statement ”$\forall x$, HasMoney(x)” says that for every person x, they have money. On the other hand, the statement ”$\exists x$, HasMoney(x)” says that there exists a person x such that they have money. We can express therefore complex relationships succinctly and clearly using predicates and quantifiers, allowing us to reason about these relationships in a systematic way.

2.2 Modal logic

Modal logic is a sophisticated extension of propositional and predicate logic and is designed to handle different modalities of truth. It offers a useful alternative to predicate logic by providing a more natural and intuitive approach to handling different types of truth. In addition, modal logics often have efficient logical processes, making them decidable and programmable in comparison with predicate logic.

The extension brought by modal logic is adding modal operators that allow us to reason about the statement’s different types of truth. The modal operators typically used are ”necessarily” and ”possibly”, denoted by the symbols ”$\Box$” and ”$\Diamond$” respectively. “ They are called ”alethic” modalities, from the Greek word for truth. In traditional terminology, Necessarily P is an ”apodeictic judgment,” Possibly P a ”problematic judgment,” and P, by itself, an ”assertoric judgment.” ”[4]. Note that this is a multi-modal logic since we are considering two different types of modalities. Anyhow Kracht[6] shows that multi-modal logic can still be simulated by mono-modal logic and we will understand why box and diamond are dual by the end of this section.

There are several varieties of modal logics, and these represent what modern approaches to grammar put together as adverbials, that is, as modifiers of adjectives, verbs, and, subsequently, nouns, but which classical linguists have classified as tense, mood, and aspect. Each of them is specifically designed to define and provide justification for various concepts, such as necessity, possibility, knowledge, belief, obligation, change in time, programme behaviour, etc. Some worth mentioning branches of modal logic are:

1. Epistemic modal logic: This branch of modal logic addresses the ideas of belief and knowledge. It is used to make arguments about what someone knows, what they believe to be true, and how knowledge and belief relate to one another. The modal operator ”necessarily” is used in this logic to represent knowledge, while ”possibly” is used to represent belief, i.e. $\Box A$ reads as ‘I know A’ or ‘A is known’ and $\Diamond A$ reads as ‘I believe A’.

2. Temporal modal logic: This branch of logic is concerned with the idea of time and the connections between occurrences. The modal operator ”necessarily” denotes events that are always true in all possible worlds, whereas ”possibly” denotes events that might be true in some possible worlds, i.e. $\Diamond A$ reads as ‘A will be true sometime in the future’ and $\Box A$ reads as ‘A is always true’.

3. Deontic modal logic: This branch deals with the concept of obligation and permission. It is used to reason about what is required, what is permitted, and the relationships between them. In this logic, the modal operator ”necessarily” represents obligations that must be fulfilled, while ”possibly” represents actions that are permitted but not required, i.e. $\Box A$ reads as ‘A is obligatory’ and $\Diamond A$ reads as ‘A is permitted’.

“According to Leibniz’s conception, our basic modal concepts—necessity, contingency, possibility, and impossibility—can be defined in straightforwardly non-modal terms.

a) Possibility: A proposition is possible if and only if it is true in some possible world. A being is possible if and only if it exists in some possible world.

b) Contingency: A proposition is contingently true if and only if it is true in this world and false in another world. A proposition is contingent if its contrary does not imply a contradiction.

c) Necessity: A proposition is necessarily true if and only if it is true in every possible world.

d) Impossibility: A proposition is impossible if and only if it is not true in any possible world."[9]

With that in mind, we can define the basic modal language by a set of proposition letters (or proposition symbols or propositional variables) ϕ. The well-formed formulas ϕ are hence given by the formula :

$$\phi ::= \text{prop} \mid \bot \mid \neg\phi \mid (\phi \vee \psi\ ) \mid (\phi \wedge \psi\ ) \mid (\phi \rightarrow \psi\ ) \mid (\phi \leftrightarrow \psi\ ) \mid \Diamond\phi \mid \Box\phi$$

Here *prop* ranges over all elements of ϕ. This definition means that a formula is either a proposition letter, the propositional constant falsum ('$\bot$'), all the propositions defined by the propositional logic, or a formula prefixed by a diamond or a square.[2]

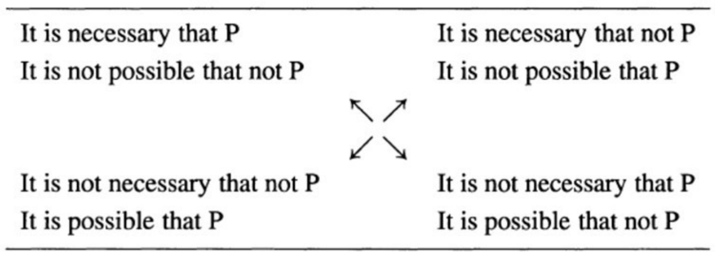
It's interesting to note that, just as the familiar first-order existential and universal quantifiers are duals to each other (i.e. $\forall$ x $\alpha \leftrightarrow \neg\exists$ x $\neg\alpha$), we have the dual operator box for our diamond.

Definition 1. $\Box\phi := \neg\Diamond\neg\phi$.

Therefore, together with the implication rule being rewritten as $\phi \rightarrow \psi := \neg\phi \vee \psi$ and the conjunction as a disjunction $\neg\phi \wedge \neg\psi := \neg(\phi \vee \psi)$ that are commonly understood, we can simplify further the rule for the basic modal language.

Definition 2. $\phi ::= \text{prop} \mid \bot \mid \neg\phi \mid (\phi \vee \psi\ ) \mid \Diamond\phi$

Aristotle's work[1] established the fundamental logical links between necessity and possibility with The Modal Square. This clearly shows the equivalences we just mentioned.



It is necessary that P It is not possible that not P		It is necessary that not P It is not possible that P
	$\nwarrow\ \nearrow$ $\swarrow\ \searrow$	
It is not necessary that not P It is possible that P		It is not necessary that P It is possible that not P

Figure 2.1: Modal Square of Opposition

Note that the schemata in each column are sub-alternatives (the top implies the bottom), the schemata in the top row are contraries (cannot both be true), the schemata in the bottom row are sub-contraries (cannot both be false), and the schemata across the diagonals are contradictories (cannot have the same truth value).[4]

Our work will revolve mainly around the simplest normal modal logic, logic K, that serves as a foundation for other modal logics. It is a formal system that uses symbols and rules of inference to reason about the truth of statements in various possible worlds. For the classical propositional logic, Frege[3] defines four main rules and axioms. Note that $\vdash$ means something is proven or is entailed.

Rule Modus ponens: $\vdash$ F and $\vdash$ F $\rightarrow$ G $\Rightarrow$ $\vdash$ G

Axiom S: $\vdash (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

Axiom K: $\vdash A \rightarrow (B \rightarrow A)$

Axiom N: $\vdash ((A \rightarrow \perp) \rightarrow \perp) \rightarrow A$

The further rules for negation, disjunction and conjunction are intuitive from the definition. To get modal propositional modal logic we add the necessitation rule and various axioms.

Necessitation Rule: $\vdash F \Rightarrow \vdash \Box F$

Normal axiom: $\vdash \Box(F \rightarrow G) \rightarrow (\Box F \rightarrow \Box G)$

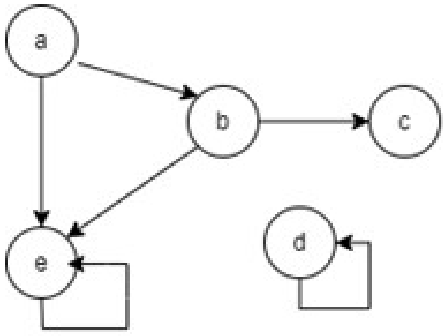
Normal model systems all contain Necessitation and Normal. The Kripke system (logic K) is therefore the least normal system, being composed of just propositional logic, necessitation and normal axiom.[5]

2.3 Kripke Semantics

Saul Kripke was a notable logician and philosopher of language who made significant contributions to several fields of logic and philosophy of language, particularly modal logic theory. He is most known for pioneering the theory of Kripke frames and Kripke models, which give a formal and logical framework for evaluating logic's concept of modality. Kripke's work has become a fundamental part of modern modal logic and has had a significant impact on various fields including philosophy, artificial intelligence, computer science, and linguistics. The standard semantics of propositional modal logic is given thus nowadays by the Kripke semantics (possible word semantics) defined in his paper from 1963 [7].

2.3.1 Kripke frames

Kripke frames also referred to as ”possible worlds models” offer a way to formally depict modal statements and their associations. They are made up of a set of possible worlds, each world representing a comprehensive and consistent set of propositions and their truth values. Additionally, Kripke frames consist of a set of accessibility relationships between the possible worlds, reflecting the relationships between the propositions. Therefore, a Kripke frame, also known as a modal frame, is a pair, $\langle W, R\rangle$, where W is a non-empty set and R is a binary relation on W. W elements are referred to as nodes or worlds, while R is referred to as the accessibility relation. Depending on the accessibility relation’s properties the relevant frame is described as transitive, reflexive, etc.



$$\mathcal{F} = \langle W, R\rangle$$

$$\text{W} = \{\text{a,b,c,d,e}\}$$

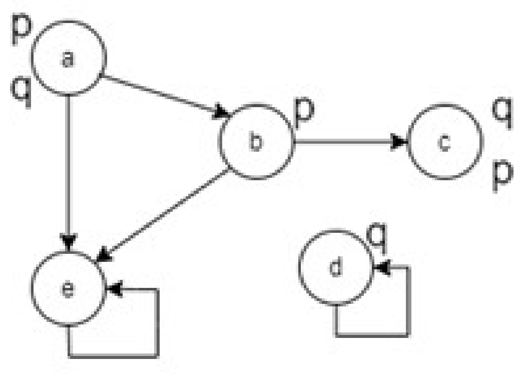
$$\text{R} = \{(\text{a,b}), (\text{a,e}), (\text{b,e}), (\text{b,c}), (\text{e,e}), (\text{d,d})\}$$

Figure 2.2: Example Kripke frame

For (a,b) $\in$ R we say that ‘a sees b’, ‘b is accessible from a’ or ’b is an alternative of a’.

2.3.2 Kripke models

Kripke models build on Kripke frames by enabling us to assign truth values to propositions in each of the possible worlds and examine the relationships between them. This allows us to make rigorous and formal inferences about the truth of modal statements, such as necessity and possibility in the actual world. Therefore, A Kripke model is a triple $\mathcal{M}$ = (W, R, $\mathcal{V}$) where $\langle W, R\rangle$ is a Kripke frame and $\mathcal{V}$ is a valuation function that maps each world in W to a set of propositional variables that are true at that world.



If p is a propositional variable, then $(\mathcal{M}$,w$) \models$ p ($\mathcal{M}$ and w satisfy/make true the formula p) if p is among the labels of w.
If p is not among the labels of w, then $(\mathcal{M}$,w$) \not\models$ p.

Figure 2.3: Example Kripke model

That being so, for every modal formula A, Kripke model $\mathcal{M}$ and world w in $\mathcal{M}$, we will define the notions:

$(\mathcal{M}$,w$) \models$ A reads "A is true in $\mathcal{M}$ at world w"

$(\mathcal{M}$,w$) \not\models$A reads "A is false in $\mathcal{M}$ at world w"

Defining modal language further, we can now understand how evaluating modal formulas works in the context of Kripke models. The following rules constitute the bases necessary for understanding the procedure presented next.

$$(\mathcal{M},\mathrm{w}) \models \neg \mathrm{A} \iff (\mathcal{M},\mathrm{w}) \not\models \mathrm{A}$$

$$(\mathcal{M},\mathrm{w}) \models \mathrm{A}\wedge\mathrm{B} \iff (\mathcal{M},\mathrm{w}) \models \mathrm{A} \text{ and } (\mathcal{M},\mathrm{w}) \models \mathrm{B}$$

For the other boolean logic connectives, the rules are defined by the truth table:

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

For evaluating modal formulas in Kripke models, we consider the following:

$$(\mathcal{M},\mathrm{w}) \models \Box\mathrm{A} \iff (\mathcal{M},u) \models \mathrm{A} \text{ for } \forall\ u \in \mathcal{M} \text{ where } \mathrm{w}\mathrm{R}u \text{ (w sees } u)$$

Therefore $\Box$A is false at a world w if there is at least one alternative world to w where A is false. $\Box$ is similar to the $\forall$ in predicate logic. One special case is the world w without alternatives, which makes $\Box$A always true at w, regardless of A.

$$(\mathcal{M},\mathrm{w}) \models \Diamond \mathrm{A} \iff \exists\ u \in \mathcal{M} \text{ such that } \mathrm{wR}u \text{ and } (\mathcal{M},u) \models \mathrm{A}$$

Therefore $\Diamond$A is false at a world w if A is false in all the worlds alternative to w. $\Diamond$ is similar to the $\exists$ in predicate logic. One special case is the world w without alternatives, which makes $\Diamond$A always false at w, regardless of A.

2.3.3 Truth, Satisfiability and Validity

Definition 3 (Truth).

$$\mathcal{M} \models \mathrm{A}$$

A formula A is true in a model $\mathcal{M}$ if $(\mathcal{M},\mathrm{w}) \models \mathrm{A}$ for all worlds w in $\mathcal{M}$ (A is true in every world of $\mathcal{M}$)

Definition 4 (Satisfiability).

$$\exists \mathrm{w} \in \mathcal{M}\ .\ (\mathcal{M},\mathrm{w}) \models \mathrm{A}$$

A formula A is satisfiable in a model $\mathcal{M}$ if there is some world w in $\mathcal{M}$ such that $(\mathcal{M},\mathrm{w}) \models \mathrm{A}$ (there exists a world w in $\mathcal{M}$ where A is true)

If A is true in $\mathcal{M}$ then A is satisfiable in $\mathcal{M}$

Definition 5 (Validity).

$$\mathcal{F} \models \mathrm{A}$$

A formula A is valid in a frame $\mathcal{F}$ if $\mathcal{M} \models \mathrm{A}$ for all models $\mathcal{M}$ based on $\mathcal{F}$
A formula A is satisfiable in a frame $\mathcal{F}$ if for some model $\mathcal{M}$ based on $\mathcal{F}$ there $\exists \mathrm{w} \in \mathcal{M}$ such that $(\mathcal{M},\mathrm{w}) \models \mathrm{A}$

Therefore A is valid in $\mathcal{F}$ iff $\neg$A is not satisfiable in $\mathcal{F}$ and A is satisfiable in $\mathcal{F}$ iff $\neg$A is not valid in $\mathcal{F}$

Validity may seem straightforward when working on a given frame, where the number of worlds and the relation between them is known. For the problem of gossip though, we might want to check the validity of a specific complex rumour in all possible words, thus determining if it is an absolute truth in all possible scenarios of knowledge and relation between individuals.

In the context of the gossip problem, worlds in a frame represent the possible combinations of which person(s) have heard which rumour(s). Formulas are then used to represent the truth or falsehood of each gossip in each of these possible worlds. So in this sense, the worlds are scenarios of what each person knows, and the formulas represent if each rumour is actually true or not in that particular world. Relations are based on communication channels between individuals, but they are not relevant for now since we are trying to find out if complex gossip formulas are valid in all combinations of possible worlds next.

2.4 Validity Proof

A proof is a way of convincing someone that a statement is true. Formally, a proof is a finite sequence of steps that follow a fixed set of rules and only refer to the structure of formulas, not their meaning. The rules that define proofs are known as a proof procedure. A proof procedure is considered sound for a particular logic if any formula that has a proof must be a valid formula in that logic, and complete if every valid formula has a proof. A proof procedure that is both sound and complete allows us to produce proofs that show that formulas are valid.

Definition 6. A WFF A of modal logic is valid iff for every $\langle W, R\rangle$, and every model (W,R,$\mathcal{V}$) based on (W,R), $\mathcal{V}$(A,w) = 1 for every w $\in$ W.

Since our focus is on the validity problem, we are looking for a proof that shows us if a given formula is valid in all frames, i.e., a universal method that gives the correct answer for all formulas. That means ($\mathcal{M}$,w) $\models$ A holds for all possible worlds in all possible models over all possible frames. There are frames with infinitely many worlds, and there are infinitely many different frames. Going for the mainstream truth table method over an infinite number of cases is not only not efficient but rather impossible. Here comes the contradiction proof.

Proof by contradiction is a powerful method of proof used in many branches of mathematics and logic, including modal logic. The basic idea behind proof by contradiction is that we assume the opposite of what we want to prove and then show that this leads to a contradiction, which in turn implies that the original assumption must be false (so our initial hypothesis must be true).

2.4.1 Tableau Algorithm

Proof by contradiction will be our useful tool in modal logic to show that a formula is valid in all possible models. This implies assuming that the formula is not valid in all frames by trying to find a counter model for the formula. This counter model should be based on a frame that has a world where the formula is false. If we find such an example, it means we proved our contradiction, so our initial formula is not valid in all frames (because we found a frame where there is a world where it is not true). We do this via a finite number of simple but exactly given steps. One thing that can happen while looking for this counter model is reaching a contradiction. Since our procedure will be defined to be sound (when not valid is returned, the formula is indeed not valid in all frames) and complete (it always finds a counter model if there is one), we can say for sure the validity of our formula even when we reach a contradiction and the formula is indeed valid.

The procedure just presented is also known as The Tableau algorithm, a validity-deciding algorithm for basic modal logic and it will be our main focus from now on once we present the ground rules for using it. Although it might seem like an efficient algorithm, its complexity grows exponentially with the number of modal operators and atomic formulas in the formula to check, hence we will also have a theoretical look at how this evolves.

Unfolding the Tableau algorithm for the K class of frames

As mentioned in Chapter 2.2, logic K is known to be the simplest form of modal logic, having no additional restriction. Now we will go through each step of the validity-deciding algorithm for this basic modal logic.

The tableau procedure is a proof method that involves constructing a proof tree for a formula by applying the following set of rules. A tableau is the equivalent of a world in our model we want to construct for the counter example, where more formulas can be either true or false. Our root node will be the tableau where our formula A is false. The proof tree is constructed by dividing the formula into sub-formulas and applying rules to each sub-formula until either a contradiction is found, or all branches of the tree have been closed. At every step, we chose a formula that hasn't been processed before and apply the corresponding rule for the outermost connective.

For the Boolean logic connectives, we unfold the formula in the same tableau, but in some cases, we have two alternative paths we need to check.

1. Negation
   - If ¬A is true then A is added as being false in that tableau
   - If ¬A is false then A is added as being true in that tableau
2. Conjunction
   - If A ∧ B is true then both A and B are added as being true in that tableau
   - If A ∧ B is false then we check the two alternatives where either A is added to the tableau as being false or B is added as being false
3. Disjunction
   - If A ∨ B is true then we check the two alternatives where either A is added to the tableau as being true or B is added as being true
   - If A ∨ B is false then both A and B are added as being false in that tableau
4. Implication
   - If A → B is true then we check the two alternatives where either A is added to the tableau as being false or B is added as being true
   - If A → B is false then A is added as being true and B is added as being false in that tableau

Note that A and B are sub-formulas, and we aim to unfold all of them until we reach the propositional variables. Those are our dead ends and if everything has been unfolded and no contradiction was made (i.e. any formula is true and false at the same time in one tableau), it means we reached a counter example model where our initial formula is not true in all words, that is it is false in our root tableau.

For the modal logic unary connectives, the rules are a little bit more special in the sense that some of them actually create new tableaux (worlds seen by the current one) or just add formulas to any existing accessible ones.

5. Necessary
    - If $\Box$A is true that means A is true in every world seen by the current one. So we don't create any new tableau but rather update any accessible ones with A added as being true. Note that before closing the current tableau, we might need to check this formula again since other rules might have created new worlds meanwhile
    - If $\Box$A is false that means that there is some world seen by the current one where A is false. So we create a new accessible tableau where A is added as being false
6. Possibly
    - If $\Diamond$A is true that means that there is some world seen by the current one where A is true. So we create a new accessible tableau where A is added as being true
    - If $\Diamond$A is false that means A is false in every world seen by the current one. So we don't create any new tableau but rather update any accessible ones with A added as being false. Note that before closing the current tableau, we might need to check this formula again since other rules might have created new worlds meanwhile

For formula $\Box(p \to q) \land \Diamond r$ for example, the algorithm works as follows:

Tableau 1:

True	False
	$\Box(p \to q) \land \Diamond r$

Applying the conjunction in the false column we get two alternatives:

Tableau 1.1:

True	False
	$\Box(p \to q)$

Tableau 1.2:

True	False
	$\Diamond r$

For Tableau 1.1 we apply the $\Box$ in the false column rule that creates a new accessible tableau from this one:

Tableau 1.1:

True	False
	$\Box(p \to q)$

$\downarrow$

Tableau 2:

True	False
	$p \to q$

There is nothing else to unfold in Tableau 1.1, so we can now close it and focus on Tableau 2, where we apply the $\to$ in the false column rule:

Tableau 2:

True	False
	$p \to q$
p	q

Both p and q are variables, so there is nothing else to unfold here either. Tableau 2 is also closed and no contradiction was found in this branch. For the $\Box(p \to q)$ formula, we can now say that it is invalid in all frames because it is invalid in the model we just created with the tableaux.

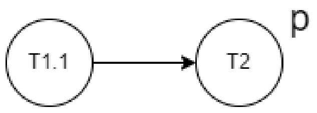


Figure 2.4: Countermodel

In Figure 2.4, $\Box(p \to q)$ is not true in world T1.1, therefore it is invalid in all frames because it is not valid in all worlds. Therefore it is not necessary that p implies q in the world that sees a world where p is true, but not q.

For our initial formula though, we still got Tableau case 1.2 to check. By applying the $\Diamond r$ is false rule in this tableau, because there are no accessible tableaux where we could add r, we do not do anything. There are no other formulas to unfold here so we can close this tableau. No contradiction was found again, so the counter model with one single world where no variable is true makes the formula not valid in all frames. The algorithm result is therefore invalid in all frames.

The problem of checking the validity of formulas in modal logic is known to be PSPACE-complete, meaning that it is one of the most difficult problems in computer science, in terms of computational complexity. PSPACE-complete problems are those that can be solved by a non-deterministic Turing machine using polynomial space and are considered to be intractable for practical purposes. This highlights the importance of continued research into the development of the field of modal logic and algorithm design. Despite its computational complexity of $O(2^n)$, the tableau algorithm remains an important tool for exploring the intricacies of modal logic and for solving complex problems in a wide range of applications.

Chapter 3

Design

In this project, we demonstrate a Python programme that applies the tableau algorithm to do modal logic formula validity checks. A modal logic formula is entered into the programme, which then generates a tableau and checks it for closure using the rules presented. The programme outputs that the formula is valid if the tableau is closed; if not, the programme outputs that the formula is invalid and returns the counter model where there is a world where the formula is not true.

This project makes two main contributions. First, we offer a simple and effective tableau algorithm implementation for validating modal logic formulas and a user-friendly graphical interface. Second, we show off Python's strength and adaptability as a tool for formal logic programming. In the following sections, we will describe the design and implementation of our Python program for solving the tableau algorithm, as well as its evaluation and performance analysis. We will also discuss the limitations of our program and suggest directions for future work.

3.1 Design requirements

The problem statement and the project's objectives serve as the foundation for the design specifications for our suggested solution. Functional, performance and usability groups make up the requirements.

Requirement Type	**Requirements**
Functional Prerequisites	• The tableau algorithm for validating modal logic formulas must be solvable by the solution. • The program must reflect the soundness and completeness of the procedure. • The solution must be capable of handling the modal operators possibility and necessity, in addition to the known propositional logic operators. • Complex formulas with multiple variables and connectives must be handled properly by the solution.
Performance Goals	• The solution must be capable of quickly producing a tableau for a given formula, with a maximum time of 10 minutes. • The solution must be able to handle large and complex formulas without running out of memory. • The tableau's closure must be efficiently verified by the solution.
Usability	• The solution must have a user-friendly interface that allows the user to enter a formula and obtain the validity result. • The output must be clear and concise, indicating whether the formula is valid or not. • The output must contain an easy-to-read graph that represents the model that proves that the formula is not valid. • The solution must be simple to set up and operate across a range of operating systems.

These requirements will serve as a design and implementation roadmap for our solution, ensuring that it satisfies both user needs and project objectives. We will outline the design strategy and system architecture that we will employ to meet these requirements in the sections that follow. In addition, we'll go over the evaluation and testing procedures we'll use to evaluate the solution's performance as well as the algorithms and data structures we'll employ to put it into practice.

3.2 Design approach and system architecture

Our design approach for solving the tableau algorithm for validity checking of modal logic formulas is based on a modular and object-oriented architecture. This consists of three main modules: graphical user interface, lexing and parsing the input, and tableau algorithm generation and checking.

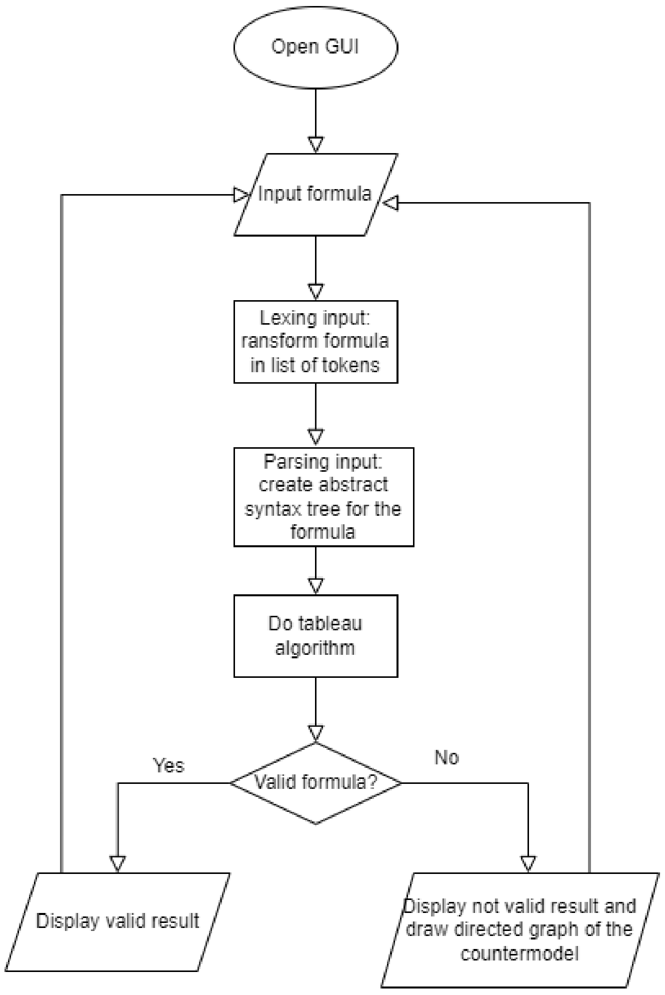

Open GUI
Input formula
Lexing input: ransform formula in list of tokens
Parsing input: create abstract syntax tree for the formula
Do tableau algorithm
Valid formula?
Yes
No
Display valid result
Display not valid result and draw directed graph of the countermodel

Figure 3.1: Tableau algorithm flowchart

3.2.1 Lexer

The Lexer class is responsible for tokenizing the input formula. It takes a string representation of the formula as input and returns a list of tokens that can be understood by the Parser. The lexer uses regular expressions to identify and extract the various components of the formula, such as atoms, variables with string or just character names, modal operators, logical connectives and parenthesis for the structure of more complex formulas. It returns the list of all the tokens in the input string in the correct order. For flexibility in writing the input formula, our lexer identifies different notations for each operator and connective. That is negation ('∼','¬','!'), conjunction ('∧','^','&'), disjunction ('|', '∨','v','V'), implication ('-⟩',→,'⇒'), necessity ('□','$_\square$','[ ]'), possibility (' ⋄','◊','⟨⟩').

For example string, for input '□(p → q) ∧ ◊r' the lexer returns the list [Token(NECESSARILY, □), Token(LPAREN, ( ), Token(VARIABLE, p), Token(IMPLIES, →), Token(VARIABLE, q), Token(RPAREN,)), Token(AND, ∧), Token(POSSIBLY, ◊), Token(VARIABLE, r)], where Token is a special class that stores the type and the symbol. The symbol is the same for the connectives but for variables, it stores the variable name.

3.2.2 Parser

The Parser class is responsible for parsing the list of tokens into a format that can be understood by the solver. The parser uses a recursive descent parsing algorithm to build an abstract syntax tree for the logical expression. It represents the structure of the formula and provides a way to navigate all the components of the formula. The parsing functions are split into three groups: parentheses together with the unary operators and variables themselves, conjunctions and disjunctions and nevertheless implication. This helps for a neater structure.

While syntax must be correct concerning the possible tokens defined in the lexer, a second layer of syntax checking is made in the parser with regard to the correct order of the tokens, so they create a valid formula structure-wise. The tree is put together with the help of a Node structure that keeps the current type of operator and the sub-nodes creating the expression. Therefore, the parser returns the root node with the outermost connective. The tree is unfolded by the sub-nodes of each node. Note that, while binary connectives have two atomic subformulas saved in a specific node, the unary connectives make use of just one of them. The variables are the leaves of the tree so they have null sub-trees.

The parser gets a string input, lexes it, and returns a Node object. The result for the example above will be Node(AND, ∧, Node(NECESSARILY, □, None, Node(IMPLIES, →, Node(VARIABLE, p, None, None), Node(VARIABLE, q, None, None))), Node(POSSIBLY, ◊, None, Node(VARIABLE, r, None, None))). For each Node, the first two components are the specific token's information, while the third and fourth are the left and right sub-nodes.

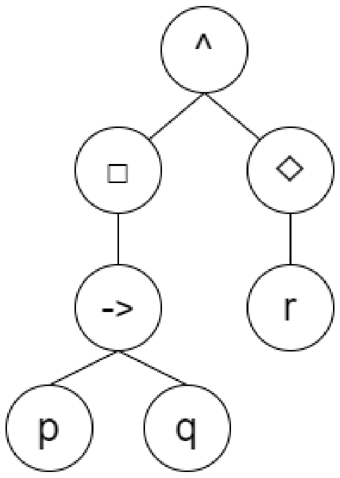


Figure 3.2: Abstract syntax tree for □(p → q) ∧ ◊r

3.2.3 Tableau Solver

The Tableau Solver takes the parse tree as input, i.e. a formula (the root node) that unfolds to other formulas (the sub-trees and leaves). Then, its scope is to check the validity of that formula by using tableaux as described in the previous chapter. This function is iterative, starting from a root tableau where the given formula is added in the false column. At the beginning of each tableau checking, we add the formulas to be unfolded to a list. We iterate over this list until it gets empty, and we can close the tableau. The appropriate rule for the outermost connective is used for every formula that hasn't been processed and the *unfolded* list is updated accordingly at each step if new formulas are added to the tableau. Depending on the rule, we may encounter:

- Adding other sub-formulas to be processed further in either the true or false column of the tableau (e.g. negation cases). Note that we save both columns as part of the tableau object in the form of a dictionary where the keys are the formulas and the values are booleans representing if the formula has been processed or not. The values are initially false but they get updated to true after processing. This prevents double-checking in the next case.

- Creating a copy tableau for checking alternative cases (e.g. conjunction as being false). This new tableau is checked in the context of a duplicate of the current Kripke model (details on this after) with the first alternative of the rule. Since it is a deep copy of the current tableau, it contains all the formulas, including the already processed ones. This is why we mentioned defining the columns as dictionaries earlier, so at the time of the initial iteration of the checking, those are not added to the *unfolded* list again. As the algorithm dictates, if a contradiction is found at any step before closing all tableaux, we can stop searching and return that our formula is valid. In the case that we reach all the leaves, we return to checking the second alternative that our rule offers. This checking is done in our original tableau while the copy one is deleted. Note that each tableau saves a list of accessible tableaux, thus when making this copy, we also make sure to make relevant copies for those seen tableaux as well. Why this is relevant we will understand in the next case.

- Saving formulas to be added in eventually accessible tableaux (e.g. a necessity in the true column). Therefore for each tableau, we define two lists: *true in accessible* and *false in accessible*. We update these lists accordingly in this case. Therefore, by the end of the iteration and before closing a tableau we can update all the accessible tableaux before checking their validity. This design choice aims to avoid the need for multiple checks on these special cases as suggested by the algorithm definition.

- Creating a new tableau seen by the current one (e.g. a possibility in the true column). The new tableau is added to the *accessible* list and it contains the formula that the rule provides.

The validity check function makes sure after every unfolded formula that there is no contradiction. This would give a valid result and it would stop the iteration. The design chooses a breadth-first search (BFS) approach for the tableaux tree. Hence after closing each tableau, the next ones to be checked are its children (i.e. accessible tableaux).

This is the structure of the tableau algorithm that provides a simple Valid or Not Valid result. We must also consider providing the counter model for the invalid cases. For this, we make use of two more classes, *Kripke world* and *Kripke model*. Each model contains a list of words and a dictionary to represent the accessibility relations. The keys represent world names, and the values are lists of other world names that are visible from the key world. Each world object gets a name and a list of variables that are true in that world, with respect to the model it is part of.

Having defined this, we can now correlate the tableaux with worlds in a model. Therefore, when initialising a new tableau, a new world with a unique name is linked to it. Checking the validity of such tableau works only given a Kripke model as an argument and this decision is motivated by the cases presented above. The work we do with the *accessible* list can be easily translated to the model by adding the world linked with each seen tableau to it and appending the right accessibility relation between them. The special case comes when working with copies of the tableau. In that case, a new model must also be created for all the copy tableaux since they are new ones with unique names. We do this by making a mapping between the names of the original tableaux and the copy ones and creating a new model where the original names are replaced with the new ones. This function is provided by the *Kripke model* class. Now every time we have to check two separate alternatives, and for the first one we duplicate the tableaux system, we also create a new model that is used for checking the validity of that specific copy system. Once a negative result is returned by all the auxiliary objects, we delete them all to free the memory. These special cases also make sure that *true and false in accessible* lists are inferred to the copies to maintain their purpose.

This entire design approach makes sure that the order in which the formulas in a tableau are chosen to be processed is irrelevant and can be completely random. That is because even in the cases where accessible tableaux exist, and a rule with parallel cases comes afterwards, the BFS implementation continues to do its work. It does that without interfering with the objects and accessibility relations due to the unique copies. This also helps with having a correct model at each step, even when checking a "copy" case. The model, in that case, is not needed though because it is never returned. However, for code completeness and correctness or further research this might be handy.

Let's remember our $\Box(\mathrm{p} \rightarrow \mathrm{q}) \wedge \Diamond\mathrm{r}$ example and go through each step of the algorithm for a better understanding of the code design we just presented. Figure 3.3 illustrates the state of the system at every step of the Tableau algorithm. To properly understand the drawing, it is crucial to remember the rules outlined in Chapter 2.4.1.

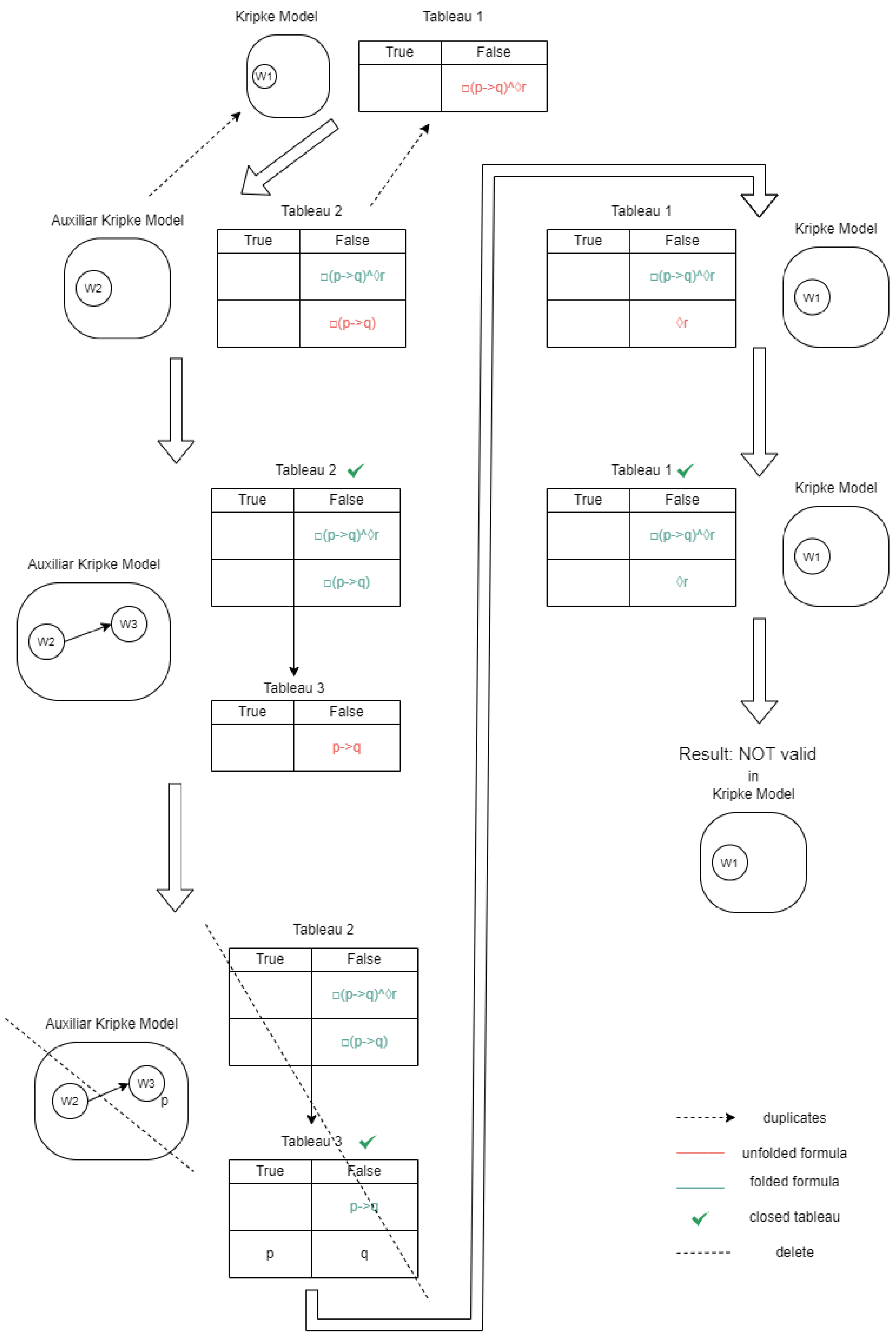
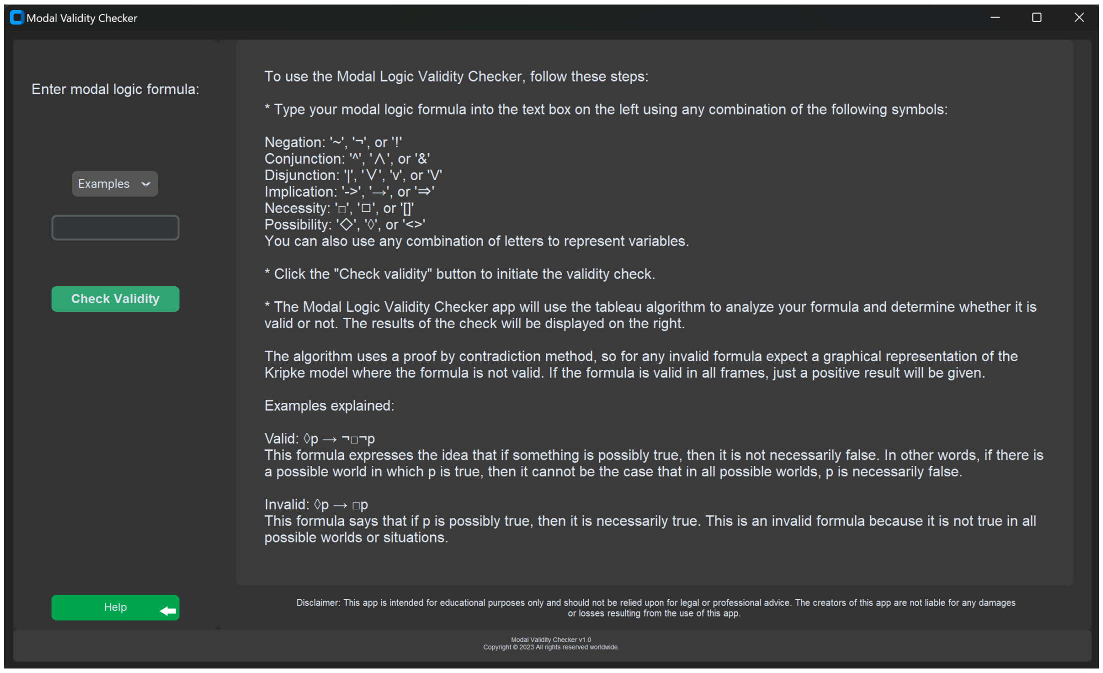


Figure 3.3: Tableau solver example

3.3 User interface design

For the simple purpose of this application, the graphical user interface (GUI) created is quite intuitive. Splitting the application screen into two columns, to the left we have the input box. There a string formula can be typed by the user and by pressing the underneath button, the result will be displayed. Helping descriptive text is also provided in the right column when starting the program and by pressing the ”Help” button at any time. After one formula is given, the right column updates by displaying the formula, the validity result (i.e. Valid or Invalid), and in the case the formula is not valid, a directed graph containing the counter model that makes the formula not valid in that frame.

The interface supports multiple usages, since after each request the input box is cleared and by typing a new one and pressing the button, the new result is displayed. For quicker use, the ”Examples” dropdown has some initial simpler and more complex formulas already written to test the program. Also, each correct input typed and checked by the user is added to that list to avoid double typing in case a formula checking should be repeated. All of this can be seen in Figures 3.4, 3.5 and 3.6.



To use the Modal Logic Validity Checker, follow these steps:

* Type your modal logic formula into the text box on the left using any combination of the following symbols:

Negation: '~', '¬', or '!'
Conjunction: '^', '∧', or '&'
Disjunction: '|', '∨', 'v', or 'V'
Implication: '->', '→', or '⇒'
Necessity: '□', '☐', or '[]'
Possibility: '◇', '◊', or '<>'
You can also use any combination of letters to represent variables.

* Click the "Check validity" button to initiate the validity check.

* The Modal Logic Validity Checker app will use the tableau algorithm to analyze your formula and determine whether it is valid or not. The results of the check will be displayed on the right.

The algorithm uses a proof by contradiction method, so for any invalid formula expect a graphical representation of the Kripke model where the formula is not valid. If the formula is valid in all frames, just a positive result will be given.

Examples explained:

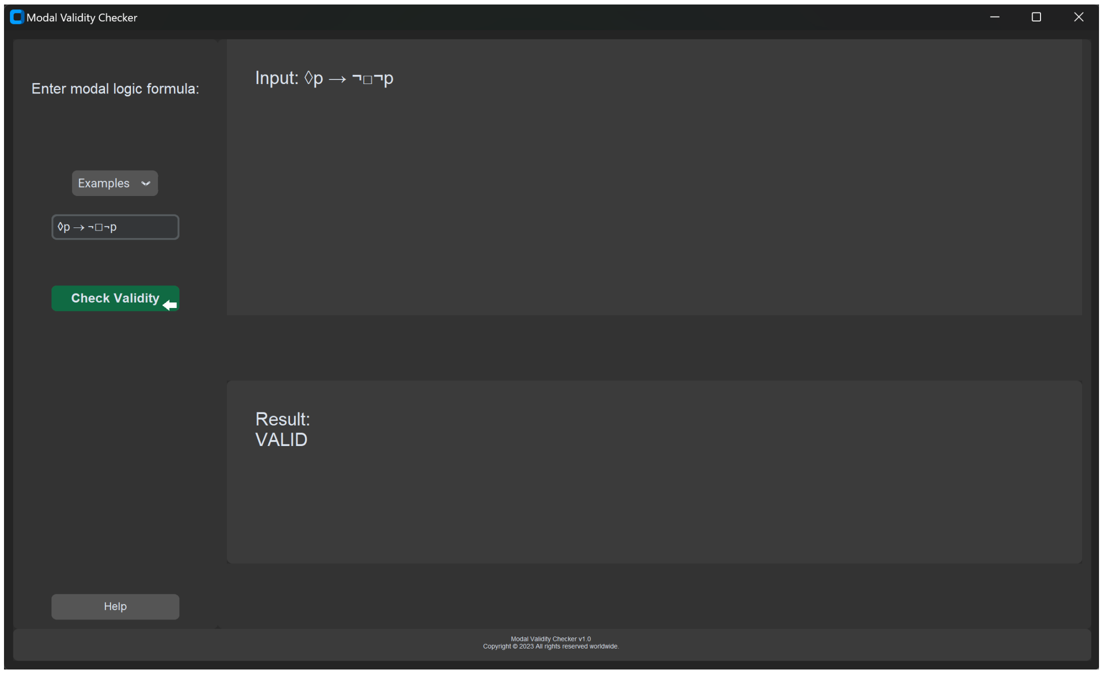
Valid: ◊p → ¬□¬p
This formula expresses the idea that if something is possibly true, then it is not necessarily false. In other words, if there is a possible world in which p is true, then it cannot be the case that in all possible worlds, p is necessarily false.

Invalid: ◊p → □p
This formula says that if p is possibly true, then it is necessarily true. This is an invalid formula because it is not true in all possible worlds or situations.

Disclaimer: This app is intended for educational purposes only and should not be relied upon for legal or professional advice. The creators of this app are not liable for any damages or losses resulting from the use of this app.

Modal Validity Checker v1.0
Copyright © 2023 All rights reserved worldwide.

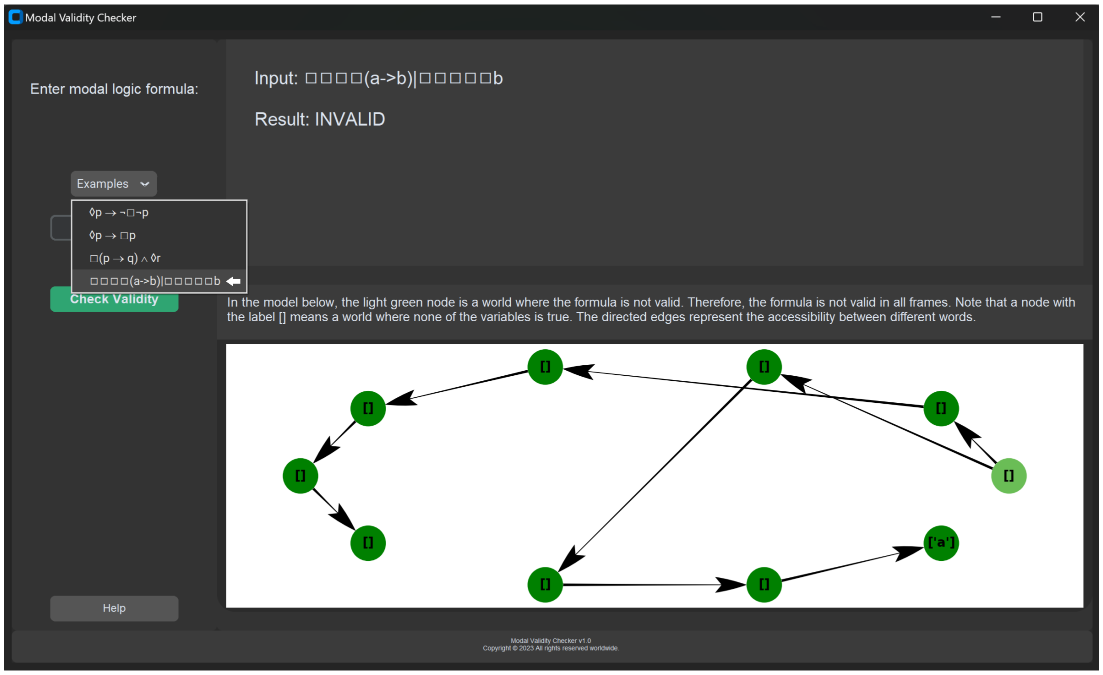
Figure 3.4: App screenshot


Modal Validity Checker
Enter modal logic formula:
Examples
◊p → ¬□¬p
Check Validity
Help
Modal Validity Checker v1.0
Copyright © 2023 All rights reserved worldwide.

Input: ◊p → ¬□¬p

Result:
VALID

Figure 3.5: App screenshot



Input: □□□□(a->b)|□□□□□b

Result: INVALID

In the model below, the light green node is a world where the formula is not valid. Therefore, the formula is not valid in all frames. Note that a node with the label [] means a world where none of the variables is true. The directed edges represent the accessibility between different words.

Figure 3.6: App screenshot

Chapter 4

Implementation

In the previous Chapters, we discussed the background and related work on modal logic and we presented the design of our Python program that performs the tableau algorithm. Now we will delve into the technical details of the implementation of the programme, having a deeper look at the code written and technical details like data structures used. We will also discuss the testing and evaluation of the program. Our goal is to provide a comprehensive understanding of the implementation of the program and its performance in various scenarios.

4.1 Programming Language and Tools

The solution was implemented in Python 3 because of its ease of use, readability, and rich set of libraries for graphical user interface development and efficiency. The following Python libraries and tools were used in our implementation:

- re: Used in the parser component to define the syntax rules for the input formula
- copy: Used for duplication of tableaux in the algorithm
- random: Used for generating complex formulas for testing purposes
- tkinter and customtkinter: Used to create a user-friendly interface for users to input modal logic formulas and receive the validity result
- networkx: Used to generate a graph of the model where the input formula is not true
- matplotlib: Used to visualize the model graph generated by networkx.

Overall, Python provided a powerful and flexible platform for developing the solution, Used for the higher-level components such as parsing and GUI, and the lower-level components such as graph generation and algorithm optimization, Python allows for the solution to handle larger and more complex problems efficiently and to provide the intuitive interface.

4.2 Components

The program implementation prioritized modularity. Thus, we will present the self-contained components of the program next. Each file is usually based around a class with a single purpose and function names are chosen to be descriptive.

4.2.1 Lexer and Parser

Parsing a text into the abstract syntax tree needed for the algorithm starts with the lexer. The Lexer class is initialised with the input text and the first character with its position. Using Token objects that keep the type and value of each recognised token, The Lexer is responsible for tokenizing the input.

Listing 4.1: Lexer tokenizer

```
def tokenize(self) -> List[Token]:
    tokens = []
    token = self.get_next_token()
    while token.type != "None":
        tokens.append(token)
        token = self.get_next_token()
    return tokens
```

The *get_next_token()* function compiles as many characters as needed for the current token to be created and it skips white spaces in the text. Using more *if* statements, it defines the modal logic language. Error handling is also covered if the syntax of the input is not valid. The entire file can be found in Appendix A.5.

The Parser class in Appendix A.4 makes use of the list of tokens generated by the Lexer and generates the tree with the help of the Node class. Each Node object has the type and value of the corresponding token, as well as two other Node objects representing the possible children for that node in the tree.

Parsing the list starts with the *parse()* function and recursively goes to *term()* and *factor()*. These three functions assure the different levels and gravities of an element in the modal formula.

The *factor* method handles the lowest-level components of the formula, such as variables, negation, modal operators and parenthesis. The children of the negation, necessary and possibly are *factor*-ed again. The content inside the parenthesis is parsed on its own and the variables are returned as leaf nodes. This complies with the fact that unary connectives are applied to any formula that is not a variable only if the formula is between parenthesis.

The *term* method handles the next level of components in the formula, such as conjunction and disjunction. It calls the *factor* method to get the left child of the new Node object it creates and then checks if the next token is a conjunction or disjunction operator. If it is, it creates a new Node object with the operator as its type and sets this left child. Afterwards, the right child is set to the result of calling the *factor* method recursively.

The *parse* method handles the highest level of components in the formula, such as the implication. The parsing starts here and goes from high to low priority of tokens for creating the final parsing tree that is returned as the root Node.

Listing 4.2: Parse method

```
def parse(self):
    node = self.term()
    if self.pos < len(self.tokens) and self.tokens[self.pos].type == "IMPLIES":
        token = self.get_next_token()
        parent = Node(token.type, token.value)
        parent.left = node
        parent.right = self.term()
        return parent
    return node
```

4.2.2 Tableau Procedure

Here we present the implementation of a Tableau algorithm for checking the validity of a given formula in a Kripke model. The implementation consists of three classes: *KripkeWorld*, *KripkeModel*, and *Tableau*.

The *KripkeWorld* has a name and a list of values(variables) associated with it. The names are usually auto-generated by a helper function as ”world x”, where x is a unique number.

KripkeModel has a list of *KripkeWorlds* as well as a dictionary representing the relations in the model by mapping each world to a list of worlds that are accessible from it. This uses the string names of the worlds instead of objects for memory benefits. One important function in the model class is *new_copy* which returns a new Kripke model that is a copy of the original, but with new world names retrieved from the mapping given as an argument. This is useful when we work with different alternatives a tableau can get.

The *Tableau* class represents a tableau in the algorithm and implements the Context Manager pattern, providing a more intuitive and safe way for users to handle resources. It has attributes for a true column, a false column, a list of unfolded propositions, a list of accessible tableaux, lists of propositions that are true or false in accessible worlds and the matching Kripke world. The class also has accessor and mutator methods. Refer to Appendix A.3 for more insight. The main focus here is on the *check_validity* method, which implements the tableau decision procedure for modal logic.

The algorithm design for the *check_validity* method is based on the tableau algorithm, which recursively builds a tableau system for the formula and determines whether it is valid or not based on the presence or absence of a contradiction. The method starts with updating the list of unfolded formulas from both columns so it can then use a *while* loop to handle the unfolding of each formula. Based on the type of the root Node of the current formula (i.e. the outermost connective), the matched rule is applied and the class variables are updated. See A.3 for a detailed implementation of each case rule. After that, it checks if there is a contradiction in the true and false columns. If there is, the formula is valid. If not, it explores the branches of the tableau by unfolding the next unexplored formula and updating the state based on the rule definition. If there are no more unexplored pones, the method checks if the formula is valid in all accessible worlds by recursively calling *check_validity* on each accessible tableau. The Kripke model object keeps track of the worlds and their relations based on the tableaux system structure. This allows for the creation of new worlds and the addition of relations between them as the tableau is unfolded.

This complex algorithm implementation is used in the user interface for each formula inputed. The invoking method that makes sure of the correct declaration of the tableau and kripke model objects at the start of the procedure is in Listing 4.3

Listing 4.3: Check validity function

```
def check_validity_of(formula):
    kripke_model= KripkeModel()
    with Tableau() as tableau:
        tableau.update_false_col_unfolded(formula)
        kripke_model.add_world(tableau.world)
        result ,model= tableau.check_validity(kripke_model)
        return result ,model
```

Efficiency and Performance

In terms of efficiency, the algorithm has a worst-case exponential time complexity, since the number of nodes in the tableau system can grow exponentially with the length of the formula. However, in practice, the algorithm is often able to prune large parts of the search space, leading to better performance for many formulas. That mostly depends on the connectives used and their frequency rather than formula length.

The implementation makes use of several Python-specific optimizations, such as the use of dictionaries to store the true and false columns of the tableau and the unfolded list, allowing for constant-time lookups, insertion and deletion operations. Copy.deepcopy is used to efficiently create copies of the tableau, while Python's built-in garbage collector is employed to automatically clean up unused objects, thus improving memory efficiency.

In terms of performance, the implementation can handle moderately complex formulas with reasonable speed but may struggle with very large or deeply nested formulas due to the $O(2^n)$ time complexity. This is dependent on several factors, including the complexity and the types of connectives that are more frequent in the formula being evaluated, and the hardware resources available. Overall though, the algorithm strikes a good balance between efficiency and readability, making use of Python's features to write a clear and concise implementation of the tableau-based algorithm for modal logic.

4.2.3 User interface

The user interface was implemented using the Python Tkinter library. The interface was designed to be simple and intuitive, and it allows the user to input a modal logic formula, initiate the validity check, and display the results of the analysis.

The *gui* file in the Appendix A.1 contains the code for the interface. It defines a class App that extends the ctk.CTk class from the customtkinter library, which provides a set of custom widgets and styles for Tkinter. The App class contains two frames: the left frame, which contains the input fields and the check validity button, and the right frame, which displays the results of the analysis. It also defines a set of methods that handle the user's interactions with the interface, such as updating the entry field with pre-defined formulas, showing the help section, and checking the validity of the formula.

The validity check is performed by calling the function in Listing 4.3 from the *tableau_procedure* module, which implements the tableau algorithm for modal logic. If the formula is valid, the interface displays a positive message. Otherwise, the interface shows a graphical representation of the Kripke model where the formula is not valid. The graphical representation is implemented in the *modelGraph* file, which defines a class GraphVisualization that extends the tk.Frame class.

To create the illustration of the Kripke model, we use the networkx library and the matplotlib library for visualization. In our implementation, we use the DiGraph() function from networkx to create the directed graph, and nx.circular_layout() to set the positions of each node in a circular layout.

To identify the special node that represents the starting tableau of the algorithm, where the input formula is put in the false column, we defined Listing 4.4.

Listing 4.4: Differentiating the node where the formula is false

```
no_incoming_edges = [node for node in G.nodes()
```

```
                    if not list(G.predecessors(node))]
node_colors = ['#6ABD56' if node in no_incoming_edges
                         else 'green' for node in G.nodes()]
```

We use this to colour the special node with a special green colour and all other nodes with green. The special node is the starting point for the tableau algorithm where the input formula is false, and it is the node that makes the formula invalid in the model and therefore in all frames.

To set the positions of each node in a circular layout, we use the code in Listing 4.5.

Listing 4.5: Design of the graph

```
pos = nx.circular_layout(G)
nx.draw(G, pos, labels=lbls, arrows=True, style='', margins=0.05,
        arrowstyle=patches.ArrowStyle('Fancy', head_length=5,
                                      head_width=2.5, tail_width=0.4),
        font_weight='bold', font_color='black', font_size=20,
        node_color=node_colors, node_size=3000,
        edge_color='black', width=6)
```

This shows how the layout and design elements of the graph are implemented. Here, *pos* is a dictionary of positions keyed by nodes. We use nx.circular_layout() to set the positions of each node in a circular layout. This creates a visually pleasing graph that shows all nodes and edges without overcrowding.

Refer to Appendix A for the entire code listing for a better understanding of this section.

4.3 Testing

Unit testing is performed on the backend using Python's built-in unittest module, resulting in a test coverage of 100%. The tests cover all major functions and edge cases, ensuring the correctness of the backend's logic. Manual testing was performed through the development stages for the frontend and integration. This testing included the functionality of all buttons, inputs, and checking correct output displays. In the next chapters, we explore edge case handling and performance metrics and how we tested that, and we discuss possible improvements.

Chapter 5

Legal, Social, Ethical and Professional Issues

In the development of any software, there are legal, ethical, social, and professional considerations to keep in mind. These considerations are particularly important for software that is designed to solve complex problems, but we will have a look at them for our modal logic validity checker as well.

5.1 Legal Issues

The implementation of the modal logic validity checker involves the use of third-party libraries. One of the key legal issues is therefore compliance with copyright and licensing laws, as the program uses external libraries that are subject to intellectual property rights. To ensure that, the program uses open-source libraries that are licensed under permissive licenses in compliance with the British Computing Society Code of Conduct and Code of Good Practice.

5.2 Social Issues

The creation of the modal logic validity checker can benefit the fields of logic and computer science from a societal standpoint. The software is an important tool for advancing knowledge and technology because it can be used to support research in academia, educational programs, and business projects. However, it's also crucial to take into account how the program might affect various demographics. The software might unfairly benefit those who have access to it, such as students at colleges with better funding or resources, if it is used in academic research or educational programs. As a result, it's crucial to think about how to make the software available to everyone, independent of their circumstances or resources.

5.3 Ethical Issues

One important ethical consideration is the potential impact of the software on individuals and society. While it may seem that a modal logic validity checker is a niche software solution, it has the potential to be used in various fields, such as in the development of safety-critical systems, which could have a significant impact on people's lives. As such, it is important to ensure that the software is designed and tested rigorously to avoid any potential harm.

5.4 Professional Issues

From a professional standpoint, the development of the modal logic validity checker requires adherence to certain standards and best practices. It was a priority to use good programming practices, such as modular and reusable code, proper documentation, and testing. It is also important to be aware of any third-party libraries used in the software and ensure that they are properly licensed and used in compliance with their terms. Additionally, we are transparent about the performance and limitations of the software. This includes providing clear documentation on the algorithm used, the input/output formats, and any potential errors or limitations of the software.

In conclusion, through the development stages of the modal logic validity checker, we considered various social, ethical, and professional issues. By keeping these considerations in mind, developers can create software that is both effective and responsible.

Chapter 6

Evaluation

This section will evaluate the solution against the functional prerequisites, performance goals, and usability requirements outlined in Chapter 3.1. Specifically, we assess the solution's ability to handle modal logic formulas with complex variables and connectives, its performance in producing tableaux and verifying their closures, and its usability in terms of user interface and output clarity. Through this evaluation, we aim to demonstrate the effectiveness of the solution in meeting the requirements of a practical and reliable modal logic validity checker.

6.1 Functionality

To meet the functional requirements, the solution implements the tableau algorithm for validating modal logic formulas in a sound and complete manner. The implementation thoroughly respects the theory in the Background Chapter, The program is capable of handling the modal operators possibility and necessity, in addition to the known propositional logic operators. The program works with satisfactory results for formulas with multiple variables and connectives, this being assured by the testing. For formulas where the result was not immediately obvious and it had a manageable length, the accuracy of the program's result was tested using a manual algorithm unfolding method performed by the author on paper.

6.2 Performance

Regarding performance goals, the solution produces a tableau for a given formula in a timely manner, with way less than the maximum time of 10 minutes, and is able to handle large and complex formulas without running out of memory. The tableau's closure is efficiently verified by the solution, as indicated by the results obtained during testing.

The test code in Appendix A.6 generates 30 formulas of varying complexity and length and measures the execution time and memory usage of the validity checker for each formula. The results are written to a CSV file, which we used to create scatter plots. These graphs show the relationship between execution time and memory usage for each formula, allowing us to evaluate the program's ability to handle large and complex formulas without running out of memory.

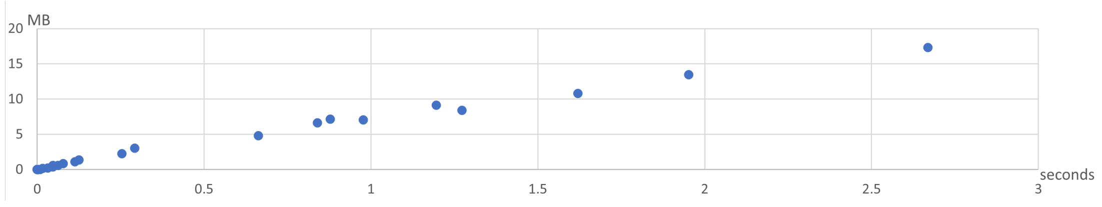


Figure 6.1: Performance with Common Connectives Frequency

In Figure 6.1 we can see a rather linear time and memory growth, these two remaining proportional for the formulas tested. The formulas generated for testing get as big as 10000 characters, so even for an input that long, the program performs within the requirements. The input values used for testing were found to be outside the bounds of typical user inputs and thus may represent an exaggerated or extreme scenario but appropriate for testing reasons.

To experiment more with the types of formulas that might get imputed in our program, we ran two separate tests with the same steps of generating the set of formulas. The difference is in the frequency of the operators. We discovered something interesting.

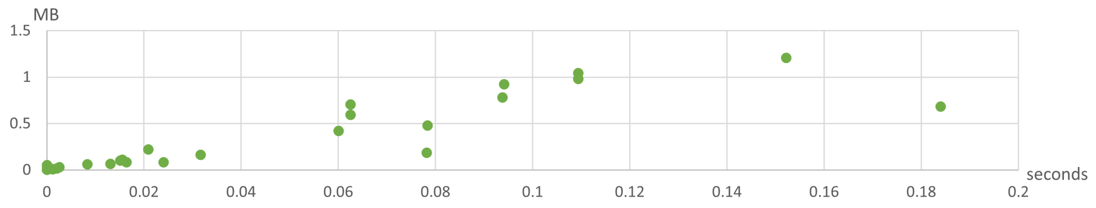


Figure 6.2: Performance with 3:1 Unary Operators

In formulas where unary connectives are more repeated, both memory usage and time complexity get smaller in value even for the very long and complex examples. See Figure 6.2. This test includes frequent repetition of the modal operators necessary and possibly. Even though at first sight, those might look like complex rules to handle by our tableau, they are actually not the ones consuming a lot of time or memory.

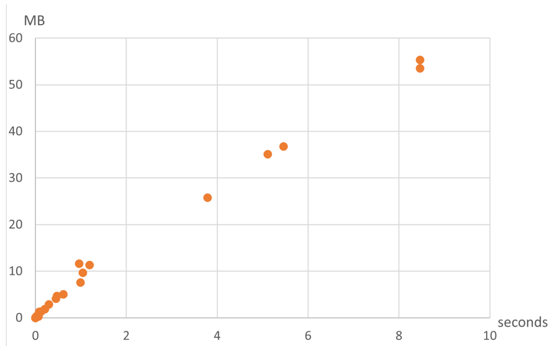


Figure 6.3: Performance with 3:1 Binary Operators

Binary connectives are the ones pressuring the performance. A reason might be the rules where two different alternatives need to be checked, so the resources might end up doubling in that cases. As shown in Figure 6.3, memory usage increases in the case where the binary operators had priority when creating the formulas. As for time for execution, we still get under 10 seconds results for the extreme scenarios.

As we mentioned in the previous chapters, this tableau validity checker algorithm has exponential worst-case time complexity. The tests conducted and their graphs from this chapter show how in practice the results are rather linear. That is not out of the perfection of our solution. We noticed that even when testing with pages long formulas, the time of execution was sometimes too good to be true and it never got too big. Most of the long formulas generated (up to 10000 in length) were getting a valid result. Therefore the lack of invalid formulas of such length might be a testing metric flaw.

Nevertheless, the program's performance with realistic inputs, both valid and invalid formulas, was well within the requirements.

6.3 Usability

To meet the usability requirements, the solution has a user-friendly interface that allows the user to enter a formula and obtain the validity result. Chapter 3.3 proves that the output is clear and concise, indicating whether the formula is valid or not. To enhance the user's understanding of the validity result, an easy-to-read graph is included that represents the model that proves that the formula is invalid. Some extreme examples from the section before were also tested in the GUI to ensure that they still display correctly and do not overflow the page. Moreover, it was noticed that some formulas take longer to be checked, but this delay is dealt with by clearing the result frame beforehand, so the user is aware that they need to wait.

Despite the potential issue with the validity of long formulas, the program still performs within the requirements for realistic inputs, both for valid and invalid formulas. This highlights the robustness and reliability of the solution in handling complex formulas, while still providing a user-friendly interface.

Chapter 7

Conclusion and Future Work

In summary, this project has showcased the effective application of modal logic and the tableau algorithm in the analysis of gossip. Through the implementation of a software solution in Python, we have demonstrated the ability to create tableaux and generate counter models for various gossip scenarios. These capabilities have allowed for the assessment of the logical soundness of gossip statements in a practical and automated manner. The algorithm serves as a valuable tool for exploring the logical properties of gossip but good algorithm design and implementation are of paramount importance.

This project has provided a comprehensive overview of the application of modal logic in understanding the logical foundations of gossip. The modal logic theories presented have demonstrated the significance of examining the logical foundations of gossip, not only in philosophy and social science but also in computer science. The theories presented may have broader applications in natural language processing, where the tools can facilitate the analysis of the certainty and possibility of statements.

Our implementation enables us to test the validity of complex gossip formulas in all conceivable world combinations. The issue of gossip necessitates determining whether a complex rumor is an absolute truth in all possible scenarios of knowledge and relation between individuals. As a result, this research has expanded our knowledge of modal logic and how it applies to gossip. The knowledge acquired from this project can be used to further research the logical foundations of gossip as well as to build more in-depth tools for analyzing information propagation in social networks.

7.1 Future work

Looking forward, several areas of future work can be pursued to build upon the findings and insights gained from this project.

Firstly, the performance of the current implementation can be further improved by optimizing the code and implementing more efficient PSPACE-complete algorithms. One potential way of improvement is the implementation of heuristics and optimizations to improve the performance of the algorithm. By prioritizing the exploration of certain branches of the tableau system and avoiding redundant computations through memoization or caching, the algorithm can be made more efficient.

Furthermore, while this project has focused on the application of modal logic in the context of gossip using the logic K, it would be worthwhile to extend this work to include other modal logics and axioms. This could provide a more nuanced understanding of the logical foundations of gossip and their applicability in different social contexts.

Finally, the current implementation is limited to the analysis of gossip in a theoretical context. Future work can focus on the practical applications of the implementation in simulating and analyzing the propagation of information. This type of application could provide valuable insights into the dynamics of societies.

Overall, this project has laid the foundation for future work in the field of gossip analysis using modal logic. By addressing some of the limitations of the current implementation and exploring new avenues for research, future work can continue to deepen our understanding of the logical foundations of gossip.

7.2 Reflections

In conclusion, the insights gained from this project provide a sobering reminder of the unreliability of gossip. The use of modal logic and the tableau algorithm have revealed that the validity of gossip statements can be a complex and difficult problem to solve. Therefore, we urge readers to exercise caution when considering the truth of any rumor or piece of gossip, as even seemingly innocuous statements can have hidden complexities and uncertainties that are not immediately apparent. So next time you hear a juicy piece of gossip, remember the lessons learned from this project and think twice before putting your trust in it.

References

[1] Aristotle. *De Interpretatione.* Clarendon Press, Oxford, UK, 1995.

[2] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic.* Cambridge University Press, Cambridge, UK, 2001.

[3] Gottlob Frege. Begriffsschrift. In Jean Van Heijenoort, editor, *From Frege to Gödel*, pages 1–83. Harvard University Press, Cambridge, 1967.

[4] Jaakko Hintikka, Johan van Benthem, and Donald Davidson, editors. *First-Order Modal Logic.* Springer Netherlands, Dordrecht, 1999.

[5] Herman Ruge Jervell. *Modal Logic.* Logos Verlag Berlin, 2013.

[6] Marcus Kracht and Frank Wolter. Normal monomodal logics can simulate all others. *The Journal of Symbolic Logic*, 64(1):99–138, 1999.

[7] Saul A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(1):67–96, 1963.

[8] Saul A. Kripke. *Naming and Necessity: Lectures Given to the Princeton University Philosophy Colloquium.* Cambridge, MA: Harvard University Press, 1980.

[9] Brandon C. Look. Leibniz's modal metaphysics. The Stanford Encyclopedia of Philosophy (Winter 2022 Edition), 2022.

Appendices