LONDON'S GLOBAL UNIVERSITY

Propositional Modal Logic Using Tableaux Methodology For Theorem Proving

by **Marcin Cuber**

supervised by **Professor Robin Hirsch**

A report presented for the degree of
MEng Computer Science

Department of Computer Science
University College London
United Kingdom
29 April 2016

All the software produced for this project can be found in the repository: https://github.com/marcincuber/modal_logic.

Abstract. The focus of this project was to study tableaux systems for propositional modal logics. The project began with a general research surrounding the topic of modal logic. It was followed by a comprehensive literature review and survey on the complexity of different modal logics. The implemented programs are automated theorem provers for propositional modal logic K, T, KB, B, K4, S4 and S5. Each implemented solver takes in a formula as an input and verifies whether it is satisfiable or not in the specific axiom system, it is accomplished using tableaux methodologies. In the process, it was established that K4 and S4 axiom systems are more difficult problems than other ones, mainly because they include a transitive relation which pushes the complexity to PSPACE-Complete. The solution proposed for these problems was the loop checking procedure, the implementation of which guarantees termination and the programs outputting of the finite model, if one exists for the input formula. All implemented theorem provers were tested for correctness using unit tests and against Spartacus software. The results produced by the theorem provers were compared against Spartacus and were verified to be correct. The additional measure used to compare these software packages was the time taken to evaluate the input formula. Following the results, programs were critically evaluated this led to the conclusions and propositions for improvements. Lastly, the report was produced, which fully documents the research, analysis, implementations, results, and testing that took place during the project. In the end, the overall project was critically evaluated and an outline of future development work was presented.

Keywords: modal logic, automated theorem proving, tableaux proof systems, satisfiability testing, model generation.

Acknowledgements. I would like to thank my supervisor Professor Robin Hirsch without his guidance, dedicated time and continuous support this project would not have been possible. I am extremely grateful that I had a chance to meet and work for such a great supervisor.

Secondly, I am immensely thankful to my individual tutor Doctor Denise Gorse for all the help I received from her and time she dedicated during my 4 years long stay at UCL.

Last, but not least, I would like to thank my family and friends for their continuous support and encouragement. In particularly, my unforgettable grandmother, exceptional mother, and my enduring girlfriend. They always believed in me and without them I would not get that far in my life, therefore I am dedicating all my effort, work and results of this project to them.

P.S. We should never give up on our dreams and never forget that Failure Is Not An Option...

Contents

1. Introduction

1.1. **Outline of Problem.**

Automated theorem proving is a type of software or methodology that helps us find an answer to a question. Such questions are directly related to reasoning, therefore these systems need to be capable of controlling the reasoning process. In general, logic is mainly focused on formalising forms of reasoning and theorem provers are dedicated to perform these tasks. There are different types of logics. This project concentrated on propositional modal logic and the tableaux methodologies were applied to reason about different tasks.

Propositional modal logic is an extension of the well known propositional logic [Hur07]. Therefore, modal logic includes all the rules of inference and all classical tautologies [Sta01, p.3] from propositional logic. Moreover, modal logic introduces two extra unary connectives, $\Box$ and $\Diamond$. The box denotes a necessity modality while the diamond is for a possibility modality.

Automated deduction systems are linked with many current fields of research, for example artificial intelligence. In this domain complex systems are performing learning and reasoning operations, where the reasoning part is mainly the automated theorem proving process [Hol05]. Nevertheless, there are other domains which make use of deduction systems; these include hardware verification, software verification of correctness, mathematical proofs, etc.[HR99] [BS98]

In basic modal logic there are also different application domains which require different settings. For this purpose there exists a structure called the Kripke model, which contains different states also called worlds, relations between those worlds, and set of formulas related to each world with truth values assigned to each formula. For example, if the model contains two worlds w_1 and w_2, and a formula $\Diamond\phi \wedge \Box\psi$ which is true at w_1, then with the relation Rw_1w_2 it holds that ϕ, ψ are true at w_2. The notation Rw_1w_2 means that there is a relation between two distinct worlds and that world w_2 can be accessed from world w_1, but not the other way around.

Modal logic has many variations, therefore the main focus was drawn to axioms K, T, KB, B, K4, S4 and S5. Each axiom represents a different modal system that produces a Kripke model; some models include restrictions. The logic K is the weakest modal logic where the model has no restrictions. In contrast S5 is the strongest modal logic where the model is satisfying reflexive, symmetric and transitive relations; such a combination is also known as equivalence relation. A model is a labelled graph that is returned by a computer program in which an evaluated formula is true.

In general, models are created to find out whether a formula is satisfiable or valid in the specific axiom. In fact if there exists a consistent model, that means the input formula is satisfiable. One of the techniques used for this purpose is semantic tableaux. This method was first introduced in 1954 by Beth (later published in [Bet55a] [Bet55b] [Bet56]) and later in 1985 was automated [Car87]. In 1972 Melvin Fitting expanded the initial propositional tableaux system and proposed a new step by step procedure which is applied in modal logic [Fit87].

For the purpose of the project complexity results were collected which indicate that axiom systems K, T, KB, B and S5 are NP-Complete, whereas systems K4 and S4 are NPSPACE-Complete. However, via Savitch's theorem [Sav70], systems K4 and S4 are in fact PSPACE-Complete. NP-Complete problems are characterised by the fact that no fast solution to them is known and the time required to solve them increases very quickly as the size of the problem grows. On the other hand PSPACE decision problems are solvable by a Turing machine using only a polynomial amount of space. These problems are suspected to be outside the complexity classes P and NP[Sip97], which means that K4 and S4 solvers are harder problems to solve. K4 and S4 axiom systems both contain transitive relations which means that solvers for them would not guarantee to terminate. For this reason, we implemented a loop checker which enables termination in all cases such that solvers (K4 and S4) always return a finite model for a given formula, if one exists.

1.2. **Project Aims, Goals and Achievements.**

The initial objective was to conduct a survey on the complexity of different modal logics. For this purpose a comprehensive literature review took place where the main focus was on the satisfiability decision problem. Additionally, complexity results for propositional temporal logic and computation tree logic were found and presented.

The major goal of the project was to develop a theorem prover for modal logic K. The theorem prover makes use of tableaux methodologies and outputs labelled graphs in which a given formula is satisfiable. In the process, the development of the initial solver for axiom system K was expanded and solvers for modal logic T, KB, B, K4, S4 and S5 were implemented. Each solver implements a parser which is crucial to the proving process, converting the input formula into a prefix format. The implemented parser allowed us to efficiently deal with formulas and allowed us to check syntax.

Lastly, we aimed at verifying that our implementations worked well and produced correct results. This had significant importance, therefore unit testing was performed during the development phase and later solvers

were tested against Spartacus (hybrid logic solver). Tests against Spartacus software concentrated on three modal axiom systems; K, T and K4. The outputs delivered by our solvers were compared using various formulas, both satisfiable and unsatisfiable.

The comparison of solvers showed that the produced results are consistent for different input formulas in each of the three axiom systems. It is important to note that Spartacus is using a novel blocking technique and implements a number of optimisations which makes it extremely efficient and fast. It was intended to implement Spartacus as a very efficient solver which additionally achieves termination by using a novel pattern-based blocking technique. In contrast our implementations were not designed for efficiency; however, they do include a loop checker which also guarantees termination. Loop checking is an essential part of axiom systems that include transitive relations because it allows them to terminate and return a finite model if one exists for the given formula.

1.3. **Overview of Sections.**

The report consists of six sections. Section I presents the general outline of the problem and what the aims, goals, and achievements are. Section II introduces the background information, definitions, and also includes complexity results and describes the methodologies used. Section III describes the format of input formulas, parser structure, and implementations of solvers. Section IV presents the results for each solver, testing and improvements. Section V provides critical evaluation and the conclusion of the overall project. Lastly, section VI is the appendix which includes the user manual, unit testing, project plan, interim report, and source code.

2. Basics of Propositional Modal Logic

2.1. **Syntax.** [[CF08]]

The language of propositional modal logic is the extension of well known propositional logic [Hur07]. Modal logic uses the signature of propositional logic and additionally contains two extra unary connectives also known as sentential operators which are $\Box$ and $\Diamond$. The box denotes a necessity modality and the diamond a possibility modality. Before we define our modal formulas we need to remember that similarly to propositional logic we have a set of atoms of propositional letters $a, b, ..., y, z$ and that we can refer to them as atomic formulas or simply atoms.

Definition 2.1. *Modal propositional formulas are defined by*

$$\phi ::= prop \mid \neg\phi \mid (\phi \vee \phi') \mid (\phi \wedge \phi') \mid (\phi \rightarrow \phi') \mid \Diamond\phi \mid \Box\phi$$

where prop is a countably infinite set of propositions.

We also point out the duality of $\Diamond$ and $\Box$ components for which the following principles are intuitively valid: $\Diamond\phi \leftrightarrow \neg\Box\neg\phi$ and $\Box\phi \leftrightarrow \neg\Diamond\neg\phi$.

Please note that Greek letter ϕ and its variations such as ϕ' used above denote arbitrary formulas.

2.2. **Semantics.** [[CF08][BF03]]

In modal logic we can come across different application domains in which we are able to reason about the states or worlds. These worlds can have different relationships with each other. So, for example we have $\Box$ modality in which a world w can be related to a world w' (relation Rww'), if w' is possible based on information described by w. In such a setting, each of the existing worlds is characterised by a valuation function which is assigning truth values $\{\bot, \top\}$ to propositional letters.

Let's now look at models in which an atomic formula can exist in different worlds with different truth valuations.

Definition 2.2. *We denote $\mathcal{M}$ to be a model in propositional modal logic which is a triple (W, R, L), where*

- *W is a set of worlds,*
- *R is a binary relation on the worlds, $R \subseteq W \times W$,*
- *L is a labelling function $L : P \rightarrow \mathcal{P}(W)$, where L assigns to each atomic formula $p \in P$ a subset $L(p)$ of W (set of worlds at which p is true). Note that $\mathcal{P}(W)$ denotes a power set of W and is equivalent to 2^W; the set of all subsets of W.*

The above defined models $\mathcal{M}$ are widely known as Kripke models named after Saul Kripke. Kripke was the first person to introduce them in the 1950s [Fit69]. In modal logic we also have a Kripke frame concept, such frame $\mathcal{F}$ is just a pair (W, R) with the same descriptions as above but does not include the labelling function L. Further explaining the above definition, we have a world $w \in W$ and accessibility relation R between worlds in W. That is $wRw' \equiv Rww'$ or simply $(w, w') \in R$ which means that world w' can be reached or accessed from w.

To further improve our understanding of Kripke models we will define the following.

Definition 2.3. *Let $\mathcal{M}$ be a model in modal logic. Let w be a world $w \in W$ and ϕ denote a formula. We will use satisfaction relation $\vDash$ to say that formula ϕ is true in the world w, written $\mathcal{M}, w \vDash \phi$, defined as follows:*

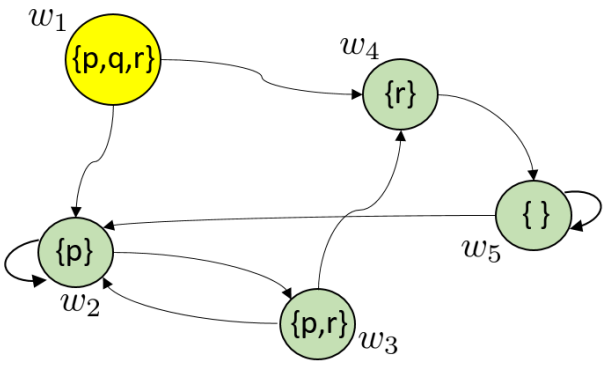
(1) $\mathcal{M}, w \vDash \top$
(2) $\mathcal{M}, w \nvDash \bot$
(3) $\mathcal{M}, w \vDash p$ *if and only if* $w \in L(p)$ *for any* $p \in P$
(4) $\mathcal{M}, w \vDash \neg\phi$ *if and only if* $\mathcal{M}, w \nvDash \phi$
(5) $\mathcal{M}, w \vDash \phi \vee \phi'$ *if and only if* $\mathcal{M}, w \vDash \phi$ *or* $\mathcal{M}, w \vDash \phi'$
(6) $\mathcal{M}, w \vDash \phi \wedge \phi'$ *if and only if* $\mathcal{M}, w \vDash \phi$ *and* $\mathcal{M}, w \vDash \phi'$
(7) $\mathcal{M}, w \vDash \phi \rightarrow \phi'$ *if and only if* $\mathcal{M}, w \vDash \phi'$ *whenever* $\mathcal{M}, w \vDash \phi$
(8) $\mathcal{M}, w \vDash \Diamond\phi$ *if and only if* $\exists w' \in W$ *such that* Rww' *and* $\mathcal{M}, w' \vDash \phi$
(9) $\mathcal{M}, w \vDash \Box\phi$ *if and only if for each* $w' \in W$ *with* Rww' *we have* $\mathcal{M}, w' \vDash \phi$

Example 2.4. *Following the definitions, we can now show an example of a Kripke model which might be easier to understand. Let's use all of the information already mentioned and let the Kripke model be $\mathcal{M} = (W, R, L)$ where*

$W = \{w_1, w_2, w_3, w_4, w_5\}$
$R = \{(w_1, w_2), (w_1, w_4), (w_2, w_2), (w_2, w_3), (w_3, w_2), (w_3, w_4), (w_5, w_2), (w_5, w_4), (w_5, w_5)\}$

$$L(i) = \begin{cases} \{w_1, w_2, w_3\}, & i = p \\ \{w_1\}, & i = q \\ \{w_1, w_3, w_4\}, & i = r \end{cases}$$

The model has the following graphical representation:



In the graph each of the relations Rw_iw_j is represented as $w_i \longrightarrow w_j$.
World w_1 is the initial node at which we start expanding our input formula and we mark it by colouring it yellow.

2.3. **Basic Concepts.**
We have defined what modal logic is and how it is linked with Kripke models. Now, we can concentrate on some basic concepts which we can refer to later on.

A formula ϕ is said to be valid if it is true at all worlds in all models, and it is satisfiable if there exists at least one model where ϕ is true. It is important to note that ϕ is valid if and only if $\neg\phi$ is not satisfiable. As we already observed the truth value of ϕ in a model $\mathcal{M}$ at world w will strictly depend on the values assigned to each atom by the L function. We will see in a later section that there are only finitely many combinations of values to consider which means that we can decide whether a formula is valid or not in a finite amount of time.

Before modal logic developed, several proof procedures for propositional logic already existed. In such procedure there is a finite number of axiom schemes in which all formulas of certain types are recognised as axioms, for instance $\neg\neg\phi \to \phi$. Also, there is a finite number of rules of inference, often just the single rule of modus ponens (implication elimination), allowing $\phi \to \psi, \phi \vdash \psi$[CH96]. More information about this will be mentioned in section 2.5.

2.4. **Filtration and Decidability.**
In order to show that validity in modal logics K, T, K4, S4 and S5 is decidable we need to introduce the concept of filtration. To find out what each letter represents see section 2.5. The filtration technique is used for proving decidability of validity by constructing finite models.

2.4.1. *Filtration.* The technique of filtration is used for proving decidability of validity by constructing finite models.

Definition 2.5. *Let $\mathcal{M} = (W, R, L)$ be a Kripke model and let Γ be a set of formulas that is closed under sub-formulas. Now, for each world $w \in W$ we define the following:*

$$\Gamma_w = \{\psi \in \Gamma : (\mathcal{M}, w) \vDash \psi\},$$

that is, Γ_w contains formulas from Γ that are true at w and consequently in model $\mathcal{M}$. Next we define an equivalence relation $\sim_\Gamma$ on worlds $w, w' \in W$,

$$\sim_\Gamma ww' \text{ is true, for all sub-formulas } \psi \in \Gamma,\ \mathcal{M}, w \vDash \psi \text{ if and only if } \mathcal{M}, w' \vDash \psi.$$

that is, worlds w and w' are equivalent if and only if they satisfy the same formulas in Γ. We also have equivalence classes $[w]$ of worlds w with respect to $\sim_\Gamma$.

Let's now consider a model with the following quotient structure $\mathcal{M}_\Gamma = (W_\Gamma, R_\Gamma, L_\Gamma)$, where

(1) $W_\Gamma = \{[w]_\Gamma : w \in W\}$.

(2) $R_\Gamma[w_0][w_1]$ *if and only if* $\exists w_0', w_1', \sim_\Gamma w_0 w_0' \wedge \sim_\Gamma w_1 w_1' \wedge R w_0' w_1'$.
(3) $L_\Gamma(p) = \{[w] : \mathcal{M}, w \vDash p\}$ *for all atomic propositions* p *in* Γ.

Such model $\mathcal{M}_\Gamma$ *is called a filtration of* $\mathcal{M}$ *through* Γ.

Additionally, as a direct consequence of point 2, we have the following:

$$if\, R_\Gamma[w][w'], \text{ then for all } \Diamond\psi \in \Gamma, \text{ if } \mathcal{M}, w' \vDash \psi, \text{ then } \mathcal{M}, w \vDash \Diamond\psi,$$

note that the choice of w *makes no difference, so this is well-defined.*

Lemma 2.6 ([Plat10, lemma 2])**.** *Let* $\mathcal{M}_\Gamma$ *be the filtration of model* $\mathcal{M}$ *through the set of propositional modal formulas* Γ *that is closed under sub-formulas. Then for all formulas* $\psi \in \Gamma$ *and worlds* $w \in W$*:*

$$\mathcal{M}_\Gamma, [w] \vDash \psi \text{ if and only if } \mathcal{M}, w \vDash \psi$$

Proof. We use a structural induction on ϕ to prove 2.6. We pick $w \in W$ and consider $[w] \in W_\Gamma$.

Base case: if ψ is an atomic formula p, $\mathcal{M}_\Gamma, [w] \vDash p$ iff $[w] \in L_\Gamma(p)$ iff (by the definition) $\mathcal{M}, w_0 \vDash p$ for some existing $\mathsf{w}_0 \in [w]$, iff $\mathcal{M}, w \vDash p$. Note that because of $[w]$ being an equivalence class all its elements agree on p.

Cases for conjunction $\wedge$ and negation $\neg$ are similar, therefore we will only consider $\wedge$. So, let's assume that $\psi \wedge \phi$ is a sub-formula of Γ. Then we have that both ψ and ϕ are also sub-formulas of Γ. Assume inductively the lemma for them. So we have, $\mathcal{M}_\Gamma, [w] \vDash \psi \wedge \phi$ iff $\mathcal{M}_\Gamma, [w] \vDash \psi$ and $\mathcal{M}_\Gamma, [w] \vDash \phi$, (by the definition 2.3) iff $\mathcal{M}, w \vDash \psi$ and $\mathcal{M}, w \vDash \phi$, iff $\mathcal{M}, w \vDash \psi \wedge \phi$ (by the induction hypothesis).

Final case: suppose that $\Box\psi$ is a sub-formula of Γ. We will assume the lemma that ψ is also a sub-formula of Γ; this way we can use the induction hypothesis. So using an assumption $\mathcal{M}_\Gamma, [w] \vDash \Box\psi$, we will prove that $\mathcal{M}, w \vDash \Box\psi$ holds. We pick another $w_1 \in W$ that satisfies relation Rww_1 and we let $[w_1]$ be the equivalence class of w_1. By the definition of R_Γ we have that $R_\Gamma[w][w_1]$. By assumption we also have that $\mathcal{M}_\Gamma, [w] \vDash \Box\psi$, so $\mathcal{M}_\Gamma, [w_1] \vDash \psi$. But $w_1 \in [w_1']$, so inductively, $\mathcal{M}, w_1 \vDash \psi$. This holds for all w_1 with relation Rww_1. So have that $\mathcal{M}, w \vDash \Box\psi$.

Now we need to prove the other direction, so we assume $\mathcal{M}, w \vDash \Box\psi$. We take $[w_1] \in W_\Gamma$ with relation $R_\Gamma[w][w_1]$. And we show the following $\mathcal{M}_\Gamma, [w_1] \vDash \psi$. By the definition of R_Γ, there exist $w' \in [w]$ and $w_1 \in [w_1]$ with relation $Rw'w_1$. Because of the equivalence relation $\sim_\Gamma w'w$ and $\Box\psi$ is a sub-formula of Γ, $\mathcal{M}, w' \vDash \Box\psi$. So $\mathcal{M}, w_1 \vDash \psi$. Inductively, we have that $\mathcal{M}_\Gamma, [w_1] \vDash \psi$. This is because $[w_1]$ was arbitrary, we also have the relation $R_\Gamma[w][w_1]$, therefore we get $\mathcal{M}_\Gamma, [w] \vDash \Box\psi$ which completes the proof. ■

Lemma 2.7 ([Plat10, lemma 3])**.** *If* Γ *in 2.6 is finite and* $\mathcal{M}$ *is an arbitrary model then* $|\mathcal{M}_\Gamma| \leq 2^{|\Gamma|}$.

Proof. There can be at most $2^{|\Gamma|}$ equivalence classes for $|\Gamma|$ formulas. ■

2.4.2. *Decidability.* Now, we have defined filtration so we can move to the concept of decidability. In order to present a clear understanding of decidability, we have the following claims:

Claim 2.8. *Given a propositional modal formula* ϕ*, the validity problem for modal languages is decidable.*

The claim states that it is possible to write a program without considering constraints of time and space which accepts a modal formula ϕ as an input, and halts (program will finish running) after a finite number of steps. Furthermore, it terminates and correctly outputs an answer telling us whether formula is valid or not.

Here we have another claim which is also related with the decidability of modal logic, specifically with satisfiability.

Claim 2.9. *In propositional modal logic a satisfiable problem is decidable. So, given a modal formula* ϕ *as an input, can we verify that* ϕ *is satisfiable in at least one model?*

Again, ignoring constraints of time and space, we can say that it is possible to write a program which accepts formula ϕ as an input, performs internal computations, and halts after a finite number of steps giving correct answer to whether ϕ is satisfiable.

Theorem 2.10. *(*[Plat10, Theorem 4]*) The validity problem in the propositional modal logic K is decidable.*

Proof. Using formula ϕ as an input, we let Γ be the set which contains all the sub-formulas of $\neg\phi$ and contains $|\Gamma|$ elements. If $\neg\phi$ is satisfiable, then by 2.7, it is also satisfiable in a Kripke model $\mathcal{M}$ which contains at most $2^{|\Gamma|}$ worlds. Therefore, a straightforward scan of all Kripke models with at most $2^{|\Gamma|}$ worlds can be used to decide whether $\neg\phi$ is satisfiable. ■

The following corollary we state without the proof. The proof is based on introducing a transitive closure; such a modification requires a new proof of lemma 2.6.

Corollary 2.11. *(*[Plat10, Corollary]*) Modal logic T, K4, S4 and S5 is decidable. Furthermore, any combinations of these axioms on top of K is decidable.*

2.5. **Modal Logic Various Axioms.**
In the previous sections we presented information which defines a basic modal logic K, which has no frame conditions. We also briefly introduced other logics. Therefore, this section will include detailed information about stronger modal logics which are built on top of logic K such that their frames include restrictions.

Although we are mainly interested in modal logics, it is important to point out that rules of inference and an axiomatisation of all tautologies [Sta01, p.3] from propositional logic also apply. The rules of inference are the basics in the construction of valid arguments. An argument is a sequence of propositions, and all apart from the last proposition are called premises since the last one is the conclusion.

Definition 2.12. *For logic K there exist two classes of axioms, which are:*

(1) *Classical tautologies- this includes all valid formulas of propositional logic.*

(2) *Scheme K- for any formula ϕ and ψ, we have that $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$.*

Definition 2.13. *Modal logic K also includes two rules of inference; an ordered pair (Γ, ϕ), where Γ is a set of formulas and ϕ is a single formula. The two rules of inference are:*

(1) *Modus Ponens:* $\dfrac{\phi \to \psi, \phi}{\psi}$

(2) *Necessitation or Generalisation:* $\dfrac{\phi}{\Box\phi}$

We now fully understand what logic K involves, therefore we can move on and present extensions of it. Other common logics, their frame conditions and complexities are listed in table 1. In addition we present axioms and their properties that need to be satisfied in the frame, these are listed in table 2.

From the information presented in the tables we can see a number of interconnections. For example, if we consider axiom K, it is part of other axioms. That means axiom such as T is a stronger system. The exact links between axioms are illustrated in figure 1 where K is the weakest and S5 is the strongest axiomatic system.

TABLE 1. Common systems of modal logic and the restrictions they place on the relations between frames and their complexity.

Axiom name	**Frame restrictions**	**Complexity**	**Result reference**
K	no restrictions	NP-Complete	[Lad77] [Hal95] [Ngu05, p.16]
T	reflexive	NP-Complete	[Lad77] [HC96, p.43] [Hal95]
D	serial	NP-Complete	[HC96, p.45] [Pri01]
K4	transitive	PSpace-Complete	[HC96, p.362] [Che80, p.131] [MO98, p.14]
KB	symmetric	NP-Complete	[Ngu05] [KPH]
B	reflexive, symmetric	NP-Complete	[HC96, p.63] [Che80, p.131]
S4	reflexive, transitive	PSpace-Complete	[Lad77] [Ngu05, p.4]
S5	reflexive, symmetric, transitive	NP-Complete	[Lad77] [MO98, p.14]
S4.3	reflexive, transitive, linear	PSpace-Complete	[Lad77] [HC96, p.362]

FIGURE 1. The relative strength of some frames in basic modal logic.

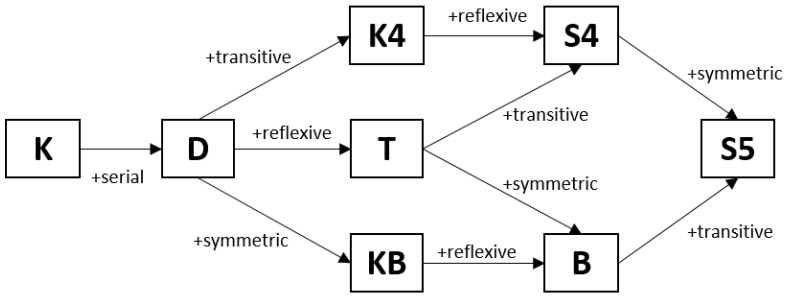


TABLE 2. Axioms and their properties forced on the frame.

Axiom name	**Axiom**
K	$\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$
D	K + $(\Box\phi \to \Diamond\phi)$
T	D + $(\Box\phi \to \phi)$
K4	K + $(\Box\phi \to \Box\Box\phi)$
KB	K + $(\Box\Diamond\phi \to \phi)$
B	T + KB
S4	T + K4 + $(\Diamond\Diamond\phi \to \Diamond\phi)$
S5	S4 + $(\Diamond\phi \to \Box\Diamond\phi)$ + $(\Diamond\Box\phi \to \Box\phi)$
S4.3	S4 + $(\Diamond\phi \wedge \Diamond\psi) \to (\Diamond(\phi \wedge \psi) \vee \Diamond(\phi \wedge \Diamond\psi) \vee \Diamond(\psi \wedge \Diamond\phi))$

In order to allow the reader to better understand relations that are applied to the Kripke frames, the following explanations illustrate exactly what is happening.

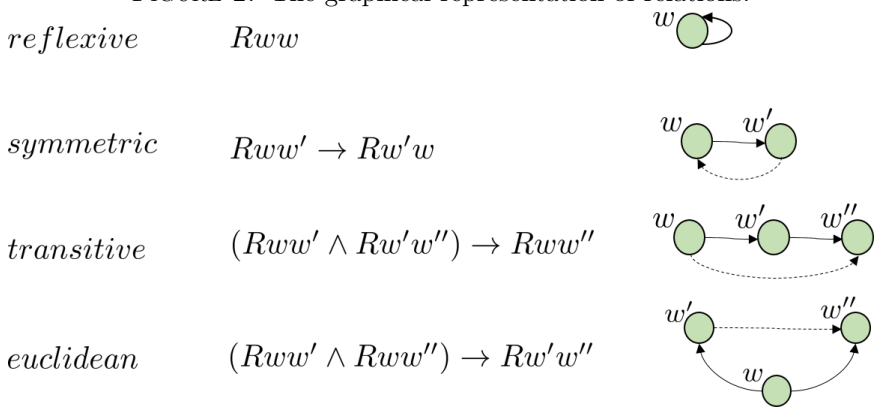
Reflexivity$(\Box\phi \to \phi)$: this axiom means that if every world $w \in W$ is accessible to itself, then any w in which formula ϕ is true will be a world from which there is an accessible world in which ϕ is true. The relationship can be expressed as Rww.

Symmetric $(\Box\Diamond\phi \to \phi)$: the accessibility relation for symmetric frames is Rww' & $Rw'w$. So, if formula ϕ is true in world $w \in W$, then in every world w' reached from w, we have a world accessible from w' at which ϕ is true.

Transitivity $(\Box\phi \to \Box\Box\phi)$: the accessibility relation for these frames can be written as: if we have worlds $w, w', w'' \in W$ with Rww' & $Rw'w''$ relations then by transitivity relation Rww'' is also true. So, $\Box\phi$ is true at w in the case that ϕ is true at every world w' with relation Rww'. Therefore, $\Box\Box\phi$ is true in all worlds accessible from every world accessible from w.

Euclidean $(\Diamond\phi \to \Box\Diamond\phi)$: this axiom was not mentioned in table 1 but we can see that in table 2 it is included as part of the S4 axiom. So, the euclidean relation does the following: $\Diamond\phi$ is true at $w \in W$ if and only if there exists world w' accessible from w where ϕ is true. Furthermore, $\Box\Diamond\phi$ is true at w if and only if for all w' accessible from w, there is another world w'' with relation $Rw'w''$ at which ϕ is true.

FIGURE 2. The graphical representation of relations.



2.6. **Frame Correspondence and Properties.**
In this section we focus on frames and properties that they have. We have already seen how different axioms are related to each other and what each axiom is.

The frame properties that we saw previously in general define a frame class. For instance, $C = \{\mathcal{F} : \mathcal{F}$ is reflexive$\}$ is the class of all reflexive frames. Now, we want to define the frame class which concerns input formulas that, if valid on a frame in a specific modal axiom, will always have the property that belongs to the frame class, e.g. C class contains reflexive frames only.

Definition 2.14. *We denote C to be a class of frames, and Σ to be a set of formulas which characterises C if and only if formula $\phi \in \Sigma$ are valid on all the frames $\mathcal{F} \in C$. So we write it as follows,*

$$\mathcal{F} \vDash \phi, \forall \phi \in \Sigma \textit{ if and only if } \mathcal{F} \in C$$

By providing the above definition, we know that every set of formulas Σ characterises a class of frames in which some relation or relations are satisfied. In previous sections we have seen different axioms which characterise different class frames. For instance, a single formula $\Box\phi \to \phi$ characterises the class of all reflexive frames and is also known as modal logic T. Another instance can be a single formula $\Box\phi \to \Box\Box\phi$ which characterises the class of all transitive frames and it is named K4.

In section 2.5 we explained in detail what reflexive, symmetric, transitive and euclidean relations are. We must also note that these axioms relate formulas to Kripke frames. They do not relate to Kripke models which are structurally triples (W, R, L), whereas frames are strictly tuples (W, R) that do not include the labelling function. In order to demonstrate that what we have in definition 2.14 is true and that axioms strictly relate to frames, not models, we provide two proofs. These proofs will concentrate on modal definability of reflexive and transitive relations.

Claim 2.15. *We begin with reflexivity, so we have that for frame $\mathcal{F}$ and formula ϕ, the following:*

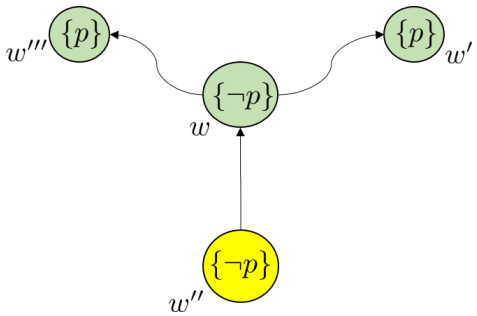
$$\mathcal{F} \vDash \Box\phi \to \phi \textit{ if and only if } \mathcal{F} \textit{ is reflexive.}$$

Proof. For the initial direction we suppose that $\mathcal{F} = (W, R)$ is reflexive. So we start with $\mathcal{F} \vDash \Box\phi \to \phi$ and will show that for all formulas ϕ, for all valuations L, and for all worlds $w \in W$, $w \vDash \Box\phi \to \phi$ holds in (W, R, L), which is our Kripke model defined as previously. Let's begin by considering a formula ϕ, valuation L, and world $w \in W$; they are all arbitrary. Now, since relation R is reflexive, Rww also holds. Suppose now $w \vDash \Box\phi$, so we have that $\forall w'$; if Rww', then $w' \vDash \phi$. And, since Rww, this means that $w \vDash \phi$ is true. This completes our initial direction and proves that $w \vDash \Box\phi \to \phi$.

Now we need to show the other direction for which we will use contraposition. We start by assuming that our frame $\mathcal{F} = (W, R)$ is irreflexive, and need to show that $\mathcal{F} \nvDash \Box\phi \to \phi$ for some formula ϕ. Basically, we are showing that if $\mathcal{F}$ is not reflexive, then there exists ϕ, valuation L, and world $w \in W$ so that $w \nvDash \Box\phi \to \phi$ holds in (W, R, L). Thus, we suppose that frame $\mathcal{F}$ is not reflexive and also we have that $w \nvDash \Box\phi \to \phi$ is equivalent to $w \vDash \Box\phi \wedge \neg\phi$. Consequently, we note that there exists a world w such that relation Rww does not hold. Next we define labelling or simply valuation function L on frame $\mathcal{F}$. So for all worlds w', we have that,

$$p \in L(w') \text{ if and only if } Rww', \text{ where p is an atomic formula.}$$

In this labelling function worlds w' are all arbitrary; however w is the world for which relation Rww does not hold. Thus, from our definition we get that $w' \vDash p$ if Rww', and for all other worlds $w'' \in W$, it implies that $w'' \nvDash p$ which is same as saying $w'' \vDash \neg p$. Let's look at the example figure below:



As previously, relation Rww does not hold, therefore $w \vDash \neg p$. However, our labelling function L implies that all worlds w' that can be reached from w, i.e. relation Rww' holds, have $w' \vDash p$. Therefore, $w \vDash \Box p$, which we can see in our figure. And also, $w \vDash \Box p \wedge \neg p$ which gives us that $w \nvDash \Box p \to p$. So, we complete our proof with the conclusion that there exists a formula $\phi = p$, where $w \nvDash \Box\phi \to \phi$. ■

Claim 2.16. *Now we consider transitivity, so we have that for frame $\mathcal{F}$ and formula ϕ, the following:*

$$\mathcal{F} \vDash \Box\phi \to \Box\Box\phi \textit{ if and only if } \mathcal{F} \textit{ is transitive.}$$

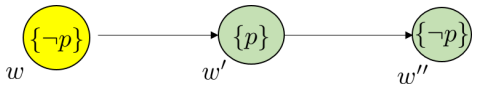
Proof. We again start with the initial direction and we suppose that $\mathcal{F} = (W, R)$ is transitive. So we start with $\mathcal{F} \vDash \Box\phi \to \Box\Box\phi$ and will show that for all formulas ϕ, for all valuations L, and for all worlds $w \in W$, $w \vDash \Box\phi \to \Box\Box\phi$ holds in (W, R, L), which is our Kripke model defined as previously. We begin by considering a formula ϕ, valuation L, and world $w \in W$; they are all arbitrary. Next, we suppose that $w \vDash \Box\phi$, and we want to show that $w \vDash \Box\Box\phi$, so we have that for all w' with relation Rww', $w' \vDash \Box\phi$. So, now we are going to consider w' with relation Rww', and in order to show that $w' \vDash \Box\phi$, we need to show that for all worlds w'', again there is a relation $Rw'w''$, $w'' \vDash \phi$.

So, we also consider world w'' with relation $Rw'w''$. Now, the transitivity of R gives us that relation Rww'' is also true. So because of $w \vDash \Box\phi$, we have that all successors of world w contain the ϕ formula. Also, there is a relation Rww'' which means w'' is a successor of w, therefore $w'' \vDash \phi$. We have now shown that for all worlds w'' with relation Rww'', $w'' \vDash \phi$. Hence $w' \vDash \Box\phi$, and so we have reached the end of the proof for this direction as $w' \vDash \Box\phi$ proves that $w \vDash \Box\phi \to \Box\Box\phi$.

Now we need to show the other direction for which we will use contraposition. We start by assuming that we have non transitive frame $\mathcal{F} = (W, R)$, and we need to show that $\mathcal{F} \nvDash \Box\phi \to \Box\Box\phi$ for some formula ϕ. Basically, we are going to show that if the frame is non transitive then there exists a formula ϕ, valuation function L, and world $w \in W$, with $w \nvDash \Box\phi \to \Box\Box\phi$ holding in model structure (W, R, L). Again we have a similar situation where $w \nvDash \Box\phi \to \Box\Box\phi$ is equivalent to saying $w \vDash \Box\phi \wedge \neg\Box\Box\phi$. So, considering a non transitive frame $\mathcal{F}$, then there are at least three worlds such as w, w', w'' with relations Rww', $Rw'w''$, and not with relation Rww''. Similarly to our previous proof, we define a valuation function on our frame, which is:

$p \in L(v)$ if and only if Rwv, where p is an atomic formula (and we used another letter for another world v).

Following our definition, we include $v \vDash p$ if there is a relation Rwv; in addition for all other worlds $w'' \in W$ we need to include $w'' \nvDash p$ or equivalently to that include $w'' \vDash \neg p$. In the following figure we can see this set-up:



Because relation Rww'' does not exist, $w'' \vDash \neg p$, which leads to $w' \vDash \neg\Box p$. Furthermore, this also means that $w \vDash \neg\Box\Box p$. Now, relating back to the definition of the valuation function, this implies that all successors of world w, these are worlds with relation Rww' have $w' \vDash p$. This means that there is $w \vDash \Box p$, hence $w \vDash \Box p \wedge \neg\Box\Box p$, and so $w \nvDash \Box p \to \Box\Box p$. Finally, there exists a formula $\phi = p$, which satisfies $w \nvDash \Box p \to \Box\Box p$. We have now completed our proof since we proved that $\Box\phi \to \Box\Box\phi$ characterises the class of all transitive frames. ■

We have now seen full proofs for modal definability of both reflexivity and transitivity. The last proposition, which we state without the proof, relates to that fact that formulas which characterise frame properties can be combined. This means that the combined sets of formulas will be characterising combined frame properties.

Proposition 2.17. *(*[Mas, p.28,Proposition 2]*) If we have a set of formulas Σ_1 that characterises frame class C_1 and another set of formulas Σ_2 that characterises frame class C_2, then combined sets i.e. $\Sigma_1 \cup \Sigma_2$ characterise a combined frame class $C_1 \cap C_2$.*

Finally, as a consequence of proposition 2.17 we have a frame class which is modally definable as well. It has the following set of formulas, $\{\Box\phi \to \Box\Box\phi, \Diamond\Box\phi \to \phi, \Box\phi \to \phi\}$, and characterises the class of equivalence frames, which is in fact axiom S5 of modal logic. The proof of the correspondence theorem for the class of reflexive, symmetric, and transitive frames is presented in [DHK07].

2.7. **Soundness and Completeness.**

In order give a clear explanation of soundness and completeness properties we need to define the following:

- A logic Σ is defined to be a set of sentences which is closed under the rules of inference that define that specific logic. Σ can be any of the axioms defined previously, e.g. K, S4, S5, etc.
- A sentence that is a member of Σ is called a theorem of that logic.
- A logic Σ is sound with respect to a class of models C only in the case when every sentence ϕ which is a theorem of Σ is valid regarding its class C.
- A logic Σ is complete regarding its class C of models when every sentence ϕ is valid with respect to C and also is a theorem of Σ.

We can interpret the above as follows: Σ-soundness: every Σ-theorem is Σ-valid, and Σ-completeness: every Σ-valid formula is a Σ-theorem. Soundness is the property that provides the initial reason for counting a logical system as desirable. In short, a formula that is provable is true. The completeness property means that every true formula is provable.

In propositional modal logic, Kripke proved the soundness and completeness of axioms K, T, K4, KB, B, S4, and S5 with respect to their specific class of models in [Kri59a] [Kri59b] [Kri63] [Kri65].

Soundness and completeness are also important properties when we consider a propositional tableau. Although the next section will describe this prove method in detail, we can still briefly present what sound and complete means. To say a tableaux method is sound, means that if we construct a closed tableau for $\neg\phi$ then ϕ must be a valid formula. Consequently, anything that is proved using tableaux methods is certainly valid. A

tableaux method is complete, meaning that if we have any valid formula ϕ then we can construct a closed tableau for $\neg\phi$. Thus, any valid formula can be proved using tableaux methods. Therefore, the tableaux approach is both sound and complete [Zal95]. The full proof of soundness and completeness of terminating modal tableaux can be found in [GHS06].

2.8. **Semantic Tableaux Theorem Proving Method.**

Semantic tableaux were first introduced in 1954 by Beth (later published in[Bet55a] [Bet55b] [Bet56]) and later in 1985 were automated[Car87]. In general this method is used for establishing the satisfiability of a formula. We only provide an input formula to the program which then tries to build a model $\mathcal{M}$ where this formula is satisfied in one of its worlds.

To create a clear view of the tableaux method, we start with propositional logic which uses it for the same purpose. A semantic tableau is a sequence of propositional formulas of the following structures:

TABLE 3. Expansion rules for propositional logic

Rule 1	**Rule 2**	**Rule 3**
$\neg(\phi \wedge \psi)$ $\neg\phi \quad \neg\psi$	$\neg(\phi \vee \psi)$ $\neg\phi$ $\neg\psi$	$(\phi \wedge \psi)$ ϕ ψ
Rule 4	**Rule 5**	**Rule 6**
$(\phi \vee \psi)$ $\phi \quad \psi$	$(\phi \rightarrow \psi)$ $\neg\phi \quad \psi$	$\neg(\phi \rightarrow \psi)$ ϕ $\neg\psi$
Rule 7	**Rule 8**	**Rule 9**
$\neg\neg\phi$ ϕ	$(\phi \leftrightarrow \psi)$ $\phi \wedge \psi \quad \neg\phi \wedge \neg\psi$	$\neg(\phi \leftrightarrow \psi)$ $\phi \wedge \neg\psi \quad \neg\phi \wedge \psi$

Tableaux approach for propositional modal logic K:

(1) tries to generate a finite tree structure satisfying w
(2) generates an initial world labelled with w, and
(3) applies tableaux rules:
   (a) propositional rules expand the label of the given world; rule for disjunction is non-deterministic.
   (b) diamond rule generates new accessible worlds
   (c) box rule extends the label of accessible worlds
   (d) clash rule detects obvious contradictions (e.g. p and $\neg p$)

2.8.1. *Tableaux by Fitting in Modal Language.* The propositional rules that we presented are part of the modal tableaux method; however the new unary connectives ($\Box$ and $\Diamond$) require further rules. In modal logic, worlds are connected by accessibility relations and these new unary connectives can express consequences about world w through a formula in the other connected world. An example to illustrate this follows: if we have a formula $\Diamond(\phi \wedge \phi')$ which is true in world w, it indicates that $(\phi \wedge \phi')$ or in fact (ϕ *and* ϕ') hold in connected world w', meaning that we have Rww'. This connection did not exist in our previous section on propositional logic, therefore none of the nine rules can be applied in this situation. In 1972 Melvin Fitting expanded the initial propositional tableaux system and proposed a new step by step procedure which takes into account connected worlds [Fit87].

Fitting's tableaux are mainly used for modal logic to prove whether a formula ϕ is valid, therefore a model for $\neg\phi$ is created. In the case where it is not possible to create a model then formula ϕ is said to be valid, otherwise

we have a counter-model. In order to deal with formulas and to move around them, prefixed formulas are used. In later sections it is demonstrated that the implemented programs actually use the same approach and that prefix notation is applied to solve problems.

The following tables present tableau rules in the prefix format. In the notation $w \vDash \psi$, w is a world which contains formula ψ.

TABLE 4. Alpha (α) and Beta (β) rules.

$\boldsymbol{\alpha}$ **formula:** $\boldsymbol{\psi}$	$\boldsymbol{\alpha_1(\psi)}$	$\boldsymbol{\alpha_2(\psi)}$	$\boldsymbol{\beta}$ **formula:** $\boldsymbol{\psi}$	$\boldsymbol{\beta_1(\psi)}$	$\boldsymbol{\beta_2(\psi)}$
$w \vDash (\phi \wedge \phi')$	$w \vDash \phi$	$w \vDash \phi'$	$w \vDash (\phi \vee \phi')$	$w \vDash \phi$	$w \vDash \phi'$
$w \vDash \neg(\phi \rightarrow \phi')$	$w \vDash \phi$	$w \vDash \neg\phi'$	$w \vDash \neg(\phi \wedge \phi')$	$w \vDash \neg\phi$	$w \vDash \neg\phi'$
$w \vDash \neg(\phi \vee \phi')$	$w \vDash \neg\phi$	$w \vDash \neg\phi'$	$w \vDash (\phi \rightarrow \phi')$	$w \vDash \neg\phi$	$w \vDash \phi'$
$w \vDash (\neg\neg\phi)$	$w \vDash \phi$	$w \vDash \phi$			

TABLE 5. Delta (δ) and Gamma (γ) rules.

$\boldsymbol{\delta}$ **formula:** $\boldsymbol{\psi}$	$\boldsymbol{\delta_1(\psi)}$	$\boldsymbol{\gamma}$ **formula:** $\boldsymbol{\psi}$	$\boldsymbol{\gamma_1(\psi)}$
$w \vDash \Diamond\phi$	$w' \vDash \phi$	$w \vDash \Box\phi$	$w' \vDash \phi$
$w \vDash \neg\Box\phi$	$w' \vDash \neg\phi$	$w \vDash \neg\Diamond\phi$	$w' \vDash \neg\phi$

Branching in this case works similarly to propositional tableaux but only in basic principles. The concept of branching takes here a different action. α formulas branch in the same way $\dfrac{\alpha}{\alpha_1, \alpha_2}$, but $\beta' s$ are managed differently. Although we have that β formulas branch into two branches, that is $\dfrac{\beta}{\beta_1 | \beta_2}$, in modal logic this generates a second graph. So, initially there is a graph G world w with a β formula. After beta expansion is applied we end up having two graphs G_1 and G_2, both having the same world but this time the formulas inside them are β_1 and β_2 respectively. δ formulas allow the graph to expand and in the case where we have $\Diamond\phi$ at world w, we create a new world w' which has binary relation Rww' such that w' contains formula (or proposition) ϕ. γ formulas act as follows: for any world that is accessible from w, that is relation Rww' exists and w contains $\Box\phi$, world w' must contain ϕ. A branch or equivalently a graph is closed if it contains $w \vDash \phi$ and $w \vDash \neg\phi$ for some $w \in W$ and ϕ. Now, if we consider a finite set of formulas Γ; we say Γ is consistent if there is no closed tableau. So, we can state without the proof the following theorem on consistency.

Theorem 2.18. *A formula ϕ is valid in logic K if and only if $\{\neg\phi\}$ is inconsistent. That is all tableaux for $\{\neg\phi\}$ are closed.*

2.9. **Related Work.**

A basic modal logic, as defined above, is a simple theoretical structure which has limited applications. This is the reason why this logic is continuously modified and studied which leads to many existing modifications. One of the ways to modify modal logic is to add more modalities which will result in multi-modal logic. Also, we can derive different types of modal logic when we add constraints on the Kripke frame. The combination of these techniques can lead us to obtaining multi-modal logics like Propositional Temporal Logic (PTL) or Computation Tree Logic (CTL). However, there are many other logics and here are some of them:

- Alethic modal logic: necessity and possibility,
  $\Box p$: it is necessary that p,
  $\Diamond p$: it is possible that p.
- Epistemic/doxastic logic: knowledge and belief of some set of agents,
  Kp: it is known that p,
  Bp: it is believed that p.
- Deontic logic: obligation, permission and prohibition,
  Op: it is obligatory that p,
  Pp: it is permitted that p,
  Fp: it is forbidden that p.

- Temporal logic: global, future and past,
  Gp: it will always be the case that p,
  Fp: it will be the case that p,
  Pp: it was the case that p.
- We also have dynamic logic, spatial logic, coalition logic, etc.

The introduction of PTL happened in 1977 and was intended for the formal verification of computer programs [Pnu77]. In general the complexity of temporal logic PTL is PSPACE-Complete. Logics such as CTL and CTL* have even higher complexity, EXPTIME-Complete and 2-EXPTIME-Complete. CTL* is an expressive branching-time temporal logic extending the standard PTL. CTL* is mainly used for developing and checking the correctness of complex reactive systems and is also used as a basis for logics for reasoning about multi-agent systems [Rey13].

3. Implementations

3.1. Section Overview.

In this section, the focus is on implementations of the theorem provers. Modal logic has many variations, therefore the implementations we will see in this section are for axiom systems K, T, KB, B, K4, S4 and S5. In order to give a clear and concise presentation of the algorithms, we provide structured pseudo code together with explanations.

Before the essence of this section is delivered, there will be information given about the programming language, libraries and data structures used. This short presentation will be followed by details about the parser that is used in our implementations. In addition, there are explanations of how input formulas (strings) need to be formatted and what the parser is returning. Next is the essence of the presentation, in particular the details of the design of the theorem provers. The final part of this section is dedicated to the implementation of axiom systems K4 and S4 because these represent a more substantial challenge. Both K4 and S4 are more difficult problems, therefore a procedure called loop checking was implemented and included as part of these algorithms. Moreover, details of the loop checking procedure are presented; this is followed by the proof of termination which is guaranteed by this procedure.

3.2. Programming Language, Libraries and Data Structures.

To implement algorithms that solve the satisfiability problem for a given model formula, Python (version 2.7.5) was used. Python is an interpreted, object-oriented programming language with dynamic semantics, and provides an extensive number of libraries.

Libraries used for the implementations:

- **NetworkX:** software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [NetX16].
- **Matplotlib:** 2D plotting library [HDFD16].

Data structures used:

- **Lists, Sets:** functions allowing to add and remove elements.
- **Dictionaries:** functions allowing adding, copying, updating and removing entries.
- **Graphs:** functions allowing adding, removing of nodes and edges, modifying node colours and labels, and copying, manipulating and displaying labelled graphs.

3.3. Parsing and String Format.

For a formula to be checked for satisfiability it first needs to be converted to a form which the program can deal with. The formula must be formatted using the following ascii data:

- **Set of propositions:** $\{a, b, c, ..., x, y, z\}$,
- **List of proposition connectives:** $[\sim, V, \hat{}, >]$, when these are parsed we get the following representations [not, or, and, imply] respectively.
- **List of modalities:** $[D, B]$, when these are parsed we get [diamond, box] respectively.

It is essential that the input formula contains the correct symbols in order for the program to parse it correctly. The parser's task is to turn the formula into prefix format, also known as tableau format as in the following examples:

Example 3.1. *Consider formula:* $\Box(p \vee q)$.
We need to write the formula in the format that the program understands so we do the following translation:

(1) *Input string for the program: B(p V q); this can also be written as (B(p V q)).*
(2) *Parser's output:* $('box', ('or', 'p', 'q'))$.

The output is in the prefix format allowing us to deal with the formula more efficiently.

Example 3.2. *Consider another formula:* $\Box(p \vee \Diamond q) \rightarrow (\Box s \wedge \neg p)$.

(1) *Input string for the program: B(p V Dq) > (Bs ^ ~p).*
(2) *Parser's output:* $('imply', ('box', ('and', 'p', ('diamond', 'q'))), ('and', ('box', 's'), ('not', 'p')))$.

Example 3.3. *The previous examples included formulas which can be successfully turned into prefix format by the parser. However, there are cases in which the parser will return an error. This can happen when a formula is ambiguous; in most cases it happens because the formula has a missing pair of brackets. Here are some examples illustrating this problem:*

(1) $\Diamond(p \vee q \wedge \Diamond s)$*: can be recognised as*
   - $\Diamond((p \vee q) \wedge \Diamond s)$ *or* $\Diamond(p \vee (q \wedge \Diamond s))$.

(2) $(p \to q \vee \Box s \to t)$*: can be recognised as*
- $((p \to q) \vee (\Box s \to t))$, $(p \to ((q \vee \Box s) \to t))$, $((p \to (q \vee \Box s)) \to t)$, *(there are few more possibilities).*

(3) $(p \to q \vee ((\Diamond s) \to t))$*: can be recognised as*
- $((p \to q) \vee ((\Diamond s) \to t))$ *or* $(p \to (q \vee ((\Diamond s) \to t)))$.

For all mentioned examples we can see that there is a possibility to derive more than one formula. This means that the original formula is ambiguous; the parser implemented for this project will in this case return an error.

In example 3.3, we have seen cases which are incorrect and in which the parser returns an error. Because the ambiguity can never occur, we know that there are a limited number of possible structures that input formulas can be transformed into. Let's consider the set of formulas $\{p, p \vee q, \Diamond p, \neg q\}$. The possible structures for each of the formulas are:

(1) Proposition: p becomes a single element in the list: ['p'].
(2) Formula: $(p \vee q)$ or more generally (p 'proposition connector' q) becomes an entry in the list which is strictly a triple: [('proposition connector', 'p' , 'q')].
(3) Formula: $(\Diamond p)$ or ('modality' p) becomes an entry in the list which is strictly a tuple: [('modality' , 'p')].
(4) Formula: $(\neg q)$ or ('negated' q) also becomes strictly a tuple: [('not' , 'q')].

These structures are the only ones that are used in parsing the string, therefore our parsered list only contains single elements, tuples or triples. It is possible that some formulas can have nested formulas inside each other, however no instance will be bigger than a triple.

3.4. **Modal Logic Algorithm Implementations.**

In order to present the algorithms in a clear and consistent format the following conventions were used:
We let G be a labelled graph with a single node n which contains formula ϕ. Additionally, $G(n)$ refers to the set of formulas $\{\phi\}$ at node n of the labelled graph G. The input formula ϕ can only be in one of the following forms:

(1) **Literal**: $(p, \neg p)$.
(2) **Alpha (α)**: $(p \wedge q), \neg(p \vee q), \neg(p \to q)$.
- If ϕ is an α formula, we let α_1, α_2 denote the two expansion formulas of ϕ. E.g, if $\phi = (p \wedge q)$ then $\alpha_1(\phi) = p$ and $\alpha_2(\phi) = q$, or if $\phi = \neg(p \vee q)$ then $\alpha_1(\phi) = \neg p$ and $\alpha_2(\phi) = \neg q$.

(3) **Beta (β)**: $(p \vee q), \neg(p \wedge q), (p \to q)$.
- If ϕ is a β formula, we let β_1, β_2 denote the two expansion formulas of ϕ. E.g, if $\phi = (p \vee q)$ then $\beta_1(\phi) = p$ and $\beta_2(\phi) = q$.

(4) **Delta (δ)**: $(\Diamond p), \neg(\Box p)$.
- If ϕ is a δ formula, we let δ_1 denote the expansion formula of ϕ. E.g, if $\phi = (\Diamond p)$ then $\delta_1(\phi) = p$.

(5) **Gamma (γ)**: $\neg(\Diamond p), (\Box p)$
- If ϕ is a γ formula, we let γ_1 denote the expansion formula of ϕ. E.g, if $\phi = (\Box p)$ then $\gamma_1(\phi) = p$.

3.4.1. *Logic K.* Propositional modal logic in its simplest form does not force any restrictions on a Kripke frame. We will present pseudo code for the logic K solver and consistency checker. The algorithm for axiom K allows us to evaluate an input formula ϕ and verify whether it is satisfiable. In the case that ϕ is satisfiable our solver will generate and return models in which the formula is always true. Each returned model is a labelled graph, and note that we can have multiple models for a given formula. In the case that there are no models we know that the formula must be unsatisfiable in modal logic K.

The following is the pseudo code illustrating the consistency check procedure for a single graph. This procedure is used to check graphs in all axiom systems excluding S5. For axiom system S5 there will be a modified version of the consistency checker which we will demonstrate later in this section.

Algorithm 1 Consistency checker

```
1: procedure Consistent(H)                    ▷ H is the graph in which we check consistency
2:     for each n ∈ H do                      ▷ go through all the nodes in the graph H
3:         if there exists a formula ψ ∈ H(n) AND ¬ψ ∈ H(n) then
4:             return No;
5:         end if
6:     end for
7:     return Yes;
```

8: **end procedure**

Algorithm 2 K- Tableaux solver

1: **procedure SATISFIABILITY**(ϕ)
2: G = new graph (ϕ) ▷ *Create graph with a single node and single formula ϕ is placed there.*
3: L = new list (G) ▷ *Initialise a list with graph G*
4: **while** L is not empty AND none of the graphs in L is completed **do**
5: H = head(L) ▷ *Select first graph from the list*
6: Let ϕ be an unticked formula in $H(n)$
7: **switch** ϕ **do**
8: **case** literal
9: tick ϕ and move to the next unticked formula
10: **end case**
11: **case** α formula
12: tick ϕ
13: replace it by $\alpha_1(\phi), \alpha_2(\phi)$
14: **end case**
15: **case** β formula
16: tick ϕ
17: let H_1 and H_2 be the two copies of H
18: add $\beta_1(\phi)$ to $H_1(n)$ and $\beta_2(\phi)$ to $H_2(n)$
19: add graphs H_1 and H_2 to L
20: delete H from list L
21: let H = head(L) ▷ *Select another graph from the beginning of the L*
22: **end case**
23: **case** δ formula
24: tick ϕ
25: add a new node n' and add new relation Rnn'
26: let $H(n') = \{\delta_1(\phi)\}$
27: **for each** predecessor m of n' **do** ▷ *for each predecessor find γ formulas and add $\gamma_1(\phi')$ to node n'*
28: **for each** ϕ' in $H(m)$ **do**
29: **if** ϕ' is a γ formula **then**
30: add $\gamma_1(\phi')$ to $H(n')$
31: **end if**
32: **end for**
33: **end for**
34: **end case**
35: **end switch**
36: **if** H is inconsistent **then** ▷ *Execute consistency checker- Algorithm 1 with input graph H*
37: Delete H from L
38: **else if** H is completed **then**
39: **return** Yes
40: **end if**
41: **end while**
42: **end procedure**

3.4.2. *Logic T.* Axiom T represents reflexive frames. For this axiom system the K-Tableaux (algorithm 2) solver was used, but required modifications which make sure that each world satisfies the reflexive relation.

Algorithm 3 T- Tableaux solver

1: **procedure SATISFIABILITY**(ϕ)
2: Execute Algorithm 2 lines 2-22 ▷ *after line 22 execute the following code*
3: **case** γ formula
4: tick ϕ

```
5:        add γ1(φ) to H(n)
6:    end case
7:    Execute Algorithm 2 lines 23-41          ▷ execute rest of the code from Algorithm 2
8: end procedure
```

Note: Algorithm 3 deals with reflexive frames; we are not adding self-loops to each of the new nodes (i.e. Rnn). These not required because the case that deals with gamma formulas ensures that the reflexivity is satisfied.

3.4.3. *Logic KB.* Axiom KB represents symmetric frames. For this axiom system, we again used the K-Tableaux procedure, but this time it required modifications which ensured that connected worlds are symmetric (see Algorithm 4).

Algorithm 4 KB- Tableaux solver

```
1: procedure SATISFIABILITY(φ)
2:    Execute Algorithm 2 lines 2-22          ▷ after line 22 execute the following code
3:    case δ formula
4:       tick φ
5:       add a new node n′ and add new relations Rnn′ and Rn′n          ▷ adding symmetric relation
6:       let H(n′) = {δ1(φ)}
7:       for each predecessor m of n′ do
8:          for each φ′ in H(m) do
9:             if φ′ is a γ formula then
10:               add γ1(φ′) to H(n′)
11:            end if
12:         end for
13:      end for
14:      for each unticked formula φ at node n′ do
15:         for each predecessor m of n′ do
16:            if φ is a γ formula then
17:               add γ1(φ) to H(m)
18:            end if
19:         end for
20:      end for
21:   end case
22:   Execute Algorithm 2 lines 35-41
23: end procedure
```

3.4.4. *Logic B.* Axiom B is the system in which frames are symmetric and reflexive. For this axiom system we combined parts of the K-Tableaux, T-Tableaux and KB-Tableaux solvers (see Algorithm 5).

Algorithm 5 B- Tableaux solver

```
1: procedure SATISFIABILITY(φ)
2:    Execute Algorithm 2 lines 2-22          ▷ expand alpha and beta formulas
3:    Execute Algorithm 3 lines 3-6           ▷ expand gamma formulas reflexivity
4:    Execute Algorithm 4 lines 3-22   ▷ expand delta formulas, line 22 of Algorithm 4 executes rest of the code
      from K-solver which completes the procedure
5: end procedure
```

3.4.5. *Logic K4.* Axiom K4 is the system in which frames satisfy transitivity. Therefore, it is a harder problem than the previous ones meaning that the approach we used previously may not guarantee termination. Thus, given an input formula, our K4 solver may end up looping infinitely. The following pseudo code is the simple implementation of the K4 solver which makes use of the initial K-Tableaux solver but additionally deals with transitive relations. The algorithm does not guarantee to terminate with a finite model for all input formulas.

Algorithm 6a K4- Tableaux solver

```
1: procedure SATISFIABILITY(φ)
2:     Execute Algorithm 2 lines 2-22                ▷ after line 22 execute the following code
3:     case δ formula
4:         tick φ
5:         add a new node n′ and add new relation Rnn′
6:         let H(n′) = {δ1(φ)}
7:         for each predecessor m of n do
8:             add mn′ to R
9:             for each φ′ in H(m) do
10:                if φ′ is a γ formula then
11:                    add γ1(φ′) to H(n′)
12:                end if
13:            end for
14:        end for
15:    end case
16:    Execute Algorithm 2 lines 35-41
17: end procedure
```

The algorithm above is a naive implementation. In order for the K4 algorithm to guarantee termination and to return a finite model (if one exists) we have implemented a loop checking procedure. The following pseudo code is the modification of the K4 solver which now includes the loop checker.

Algorithm 6b K4- Tableaux solver with loop checking

```
1: procedure SATISFIABILITY(φ)
2:     Execute Algorithm 2 lines 2-22                ▷ after line 22 execute the following code
3:     case δ formula                                ▷ implements loop checker
4:         tick φ
5:         Let S be the set of formulas
6:         include δ1(φ) in S
7:         for each predecessor m of n do
8:             for each φ′ in H(m) do
9:                 if φ′ is a γ formula then
10:                    include γ1(φ′) in S
11:                end if
12:            end for
13:        end for
14:        if there exists node k ∈ H such that H(k) == S then   ▷ check if the same node already exists in H
15:            add nk to R                           ▷ add new relation between two existing nodes
16:        else
17:            add new node n′ to H
18:            let H(n′) = S and add relation Rnn′
19:        end if
20:    end case
21:    Execute Algorithm 2 lines 35-41
22: end procedure
```

3.4.6. *Logic S4.* Axiom system S4 is a combination of the transitive axiom K4 and reflexive axiom T. So, in this system each Kripke frame is satisfying reflexive and transitive relations. Similarly to K4, the S4 algorithm will also have two forms of pseudo code; the first one (Algorithm 7a) will implement a solution which may not terminate with a finite model and the second one (Algorithm 7b) is the solution that makes use of the loop checker and guarantees to terminate with a finite model if one exists.

Algorithm 7a S4- Tableaux solver

```
1: procedure SATISFIABILITY(φ)
```

2: Execute Algorithm 2 lines 2-22 ▷ *expand alpha and beta formulas*
3: Execute Algorithm 3 lines 3-6 ▷ *expand gamma formulas reflexivity*
4: Execute Algorithm 6a lines 3-16 ▷ *line 16 of Algorithm 6a executes rest of the code from K-solver which completes the procedure*
5: **end procedure**

Algorithm 7b S4- Tableaux solver with loop checking

1: **procedure SATISFIABILITY**(ϕ)
2: Execute Algorithm 2 lines 2-22 ▷ *expand alpha and beta formulas*
3: Execute Algorithm 3 lines 3-6 ▷ *expand gamma formulas reflexivity*
4: Execute Algorithm 6b lines 3-21 ▷ *line 21 of Algorithm 6b executes rest of the code from K-solver which completes the procedure*
5: **end procedure**

Final note on the loop checking algorithm: it is only applied in the K4 and S4 algorithms. The reason is that in transitive frames we may come across cases where every time we expand a node it will contain the same gamma formulas that were already expanded. Such a problem could lead to non termination of the program. The loop checking procedure verifies whether the newly created world already exists in the graph, such that if it exists we add an edge to this node, and if not, we then create the new node for that specific delta formula.

3.4.7. *Logic S5.* This system is characterised by models where the accessibility relation is reflexive, transitive, and symmetric. In fact we call it an equivalence relation. This is an important feature because if there exists a model (graph) for a given formula, we know that it is a complete directed graph. In a directed graph every pair of distinct nodes is connected by a pair of unique edges in both directions. Since we want to present a clear and understandable model or models for each formula, the S5 algorithm omits the implicit relationships between nodes. Every time we add a new node we know that it is satisfying the equivalence relation. The elements which constitute the S5 implication are given in Algorithms 8 and 9 below.

Algorithm 8 Consistency checker- S5

1: **procedure** CONSISTENT(H, $GamS_H$) ▷ *H is a graph and $GamS_H$ is a list that contains gamma formulas which are true in all nodes*
2: **for each** $n \in H$ **do**
3: Let $H(n) = H(n) \cup GamS_H$ ▷ *merge two sets of formulas*
4: **if** there exists a formula $\psi \in H(n)$ AND $\neg\psi \in H(n)$ **then**
5: **return** No;
6: **end if**
7: **end for**
8: **return** Yes;
9: **end procedure**

Algorithm 9 S5- Tableaux solver

1: **procedure SATISFIABILITY**(ϕ)
2: G = new graph (ϕ) ▷ *Create graph with a single node and single formula ϕ is placed there.*
3: Let $GamS_G$ be a set of gamma formulas that are true in every node in G. ▷ *Initially gamma set is empty.*
4: **if** ϕ is γ formula **then** ▷ *Deal with the case where input formula is a gamma formula.*
5: tick ϕ in G ▷ *We tick ϕ since we won't be expanding it again.*
6: $GamS_G = \{\gamma_1(\phi)\}$ ▷ *Add expansion formula of ϕ to the gamma set.*
7: $G = \{\phi, \gamma_1(\phi)\}$ ▷ *Add the formula from gamma set to the single node in graph G*
8: **end if**
9: L = new list ((G, $GamS_G$)) ▷ *Initialise the list of tuples with graph (G, $GamS_G$)*
10: **while** L is not empty AND none of the graphs in L is completed **do**
11: (H, $GamS_H$) = head(L) ▷ *Select first tuple with graph H and related gamma list $GamS_H$*
12: Let ϕ be an unticked formula in $H(n)$
13: **switch** ϕ **do**
14: **case** literal

15: tick ϕ and move to the next unticked formula
16: **end case**
17: **case** α formula
18: tick ϕ
19: replace it by $\alpha_1(\phi), \alpha_2(\phi)$
20: **end case**
21: **case** β formula
22: tick ϕ
23: let H_1 and H_2 be the two copies of H
24: place $\beta_1(\phi)$ in $H_1(n)$ and $\beta_2(\phi)$ in $H_2(n)$
25: add tuples (H_1, $GamS_H$) and (H_2, $GamS_H$) to L
26: delete (H, $GamS_H$) from list L
27: let (H, $GamS_H$) = Head(L) ▷ *Pick the first tuple containing graph from the list since the current one was deleted*
28: **end case**
29: **case** δ formula
30: **if** there exists node $n' \in H$ AND $\delta_1(\phi) \in H(n')$ **then**
31: tick ϕ and go to the next unticked formula
32: **else if** $\delta_1(\phi)$ in $GamS_H$ **then**
33: tick ϕ and go to the next unticked formula
34: **else**
35: tick ϕ
36: add a new node n'' in H
37: let $H(n'') = \{\delta_1(\phi)\}$
38: let $H(n'') = H(n'') \cup GamS_H$ ▷ *include all formulas from $GamS_H$ in $H(n'')$*
39: **end if**
40: **end case**
41: **case** γ formula
42: tick ϕ
43: let $\phi' = \gamma_1(\phi)$
44: add ϕ' to $GamS_H$
45: **for each** node $n' \in H$ **do**
46: add ϕ' to $H(n')$
47: **end for**
48: **end case**
49: **end switch**
50: **if** (H, $GamS_H$) is inconsistent **then** ▷ *Execute consistency checker- Algorithm 8 with input (H, $GamS_H$)*
51: Delete (H, $GamS_H$) from L
52: **else if** H is completed **then**
53: **return** Yes
54: **end if**
55: **end while**
56: **end procedure**

3.5. **Loop Checking and Termination.**

In the previous section we have seen two algorithms for each of the axiom systems K4 (6a, 6b) and S4 (7a, 7b). We introduced versions 'b' of each because they include the loop checking procedure which guarantee termination. In this section we will discuss this procedure in more detail.

The context we are concerned with is well-defined; there are labelled graphs, expansion rules and the loop checking procedure. The loop checking only applies to systems with transitive relations, therefore we consider only the K4 and S4 systems. We already know that labelled graphs exist for a given formula only when the formula is satisfiable, and that such graphs must be consistent. The rules that are applied to the initial graph are alpha, beta, delta and gamma (see sections 2.83.4). In the process of generating finite graphs we visit every formula in every node in the graph, apply one of the rules and then tick the formula as visited. This allows us to make sure that all formulas are expanded; if all formulas in the graph are ticked and the graph is consistent then we have a

finite model. Moreover, because the formulas are of a finite length we know that we will reach a stage at which all formulas are expanded and ticked.

Now, considering that we expand a formula only once, we also know that only sub-formulas of the visited formula are added to nodes in the action. So, if we have a delta formula, the delta rule is applied which creates a new node. All the new nodes are strictly successors of the previous node, which ensures a strict decrease of the size of the formula. However, because of the loop checker, before we add a new node we first check for existence of the same formulas in any existing node in the graph. In the case that the same node already exists we do not add a new node but we instead add an edge to that ancestor.

Definition 3.4. *Looping Tableau. We call a tableau or a labelled graph looping if and only if there exists a node to which, when we apply the delta rule, we loop back to that node or we go to another existing node in the graph. Note that these looping tableaux are finite because of the loop checking procedure which either allows for the new node to be created or we tick the evaluated formula and computation is finished for that formula.*

Corollary 3.5 ([GHS06, corollary 23])**.** *Given input formula ϕ, if it has an open looping tableau in the axiom system K4 or S4 then it has a K4 model or S4 model. Tableau is said to be open if there is no node with a set of formulas that contains ψ and $\neg\psi$.*

Proposition 3.6 ([GHS06, proposition 25])**.** *We denote ϕ to be a modal formula, we present the following two implications:*

(1) *If ϕ is satisfiable in axiom K4 or S4 then there is an infinite open tableau for ϕ. This is the case in algorithms 6a and 7a which do not include loop checking.*
(2) *If there exists an infinite open tableau for ϕ then in axiom K4 or S4 there is a finite open tableau for ϕ. This is the case in algorithms 6b and 7b which include loop checking.*

Proof.

(1) The proof for this implication and details can be found in [CCGH97].
(2) For this implication we know that using the loop checker we stop the computation if all the formulas are ticked. So, the resulting model that we get is in fact an infinite tableau and it can not be closed.

■

Theorem 3.7. *The construction of tableaux for a given formula terminates and returns a finite graph if one exists. In other words when we expand a formula in K4 or S4 axiom system it will lead to a looping tableau.*

Proof. We start by making two observations related to sub-formula conditions which we already mentioned:

(1) for every graph and every node, the set of formulas at a node only includes finite number of formulas,
(2) we can only have a finite number of nodes in the graph where each node has a unique set of formulas; that is there are no nodes with the same set of formulas.

Now, if we are given a formula and we are constructing a tableau for it, we know that a new node is only created when a node with the given set of formulas does not exist in the graph. This is always true since the loop checker verifies that before the node is created. Another important fact is that we never delete formulas but we tick them once they expanded. This means that we cannot have a situation where an infinite number of nodes will contain unique sets of formulas. The only potential situation where infinite expansion of the node w could occur is when we would be introducing an infinite number of new nodes where $w \vDash \Diamond\phi$. However, since we already have a node w, the number of $\Diamond$ (delta) formulas is bounded by the linear number of sub-formulas of the input formula. Moreover, each node in the graph can only have a linear number of successors. Therefore, we have reached the conclusion that the expansion of the graph is bounded. ■

4. Results and Testing

4.1. **Section Overview.**

In this section we will focus on the results that our algorithms return. Later we will test our implementations against another tableaux solver to verify whether our results are consistent but in addition to compare the execution time. Lastly, we will have a section summarising our findings followed by a critical evaluation.

In the previous section we have presented pseudo code for each algorithm that we developed. As part of each implementation we also included consistency checkers and, where necessary, a loop checker. Since we have different programs for each axiomatic system, for each axiom there will be multiple examples displayed. We will use a range of formulas which are satisfiable but we will also present formulas which are valid, such that their negation should be unsatisfiable. We will observe that every satisfiable formula has at least one model and every unsatisfiable formula does not have a model.

Regarding the structure of our graphs, in order to improve the readability different layouts are sometimes used. We included non trivial formulas which differ in size, and note that lengthy formulas could lead to long labels and a large number of nodes. However, if we use the actual python implementation on our machine we will get dynamic figures which will allow us to move around the graph and options such as zoom in/out will be possible. These features will allow us to closely examine every model we get and verify all the details that we require.

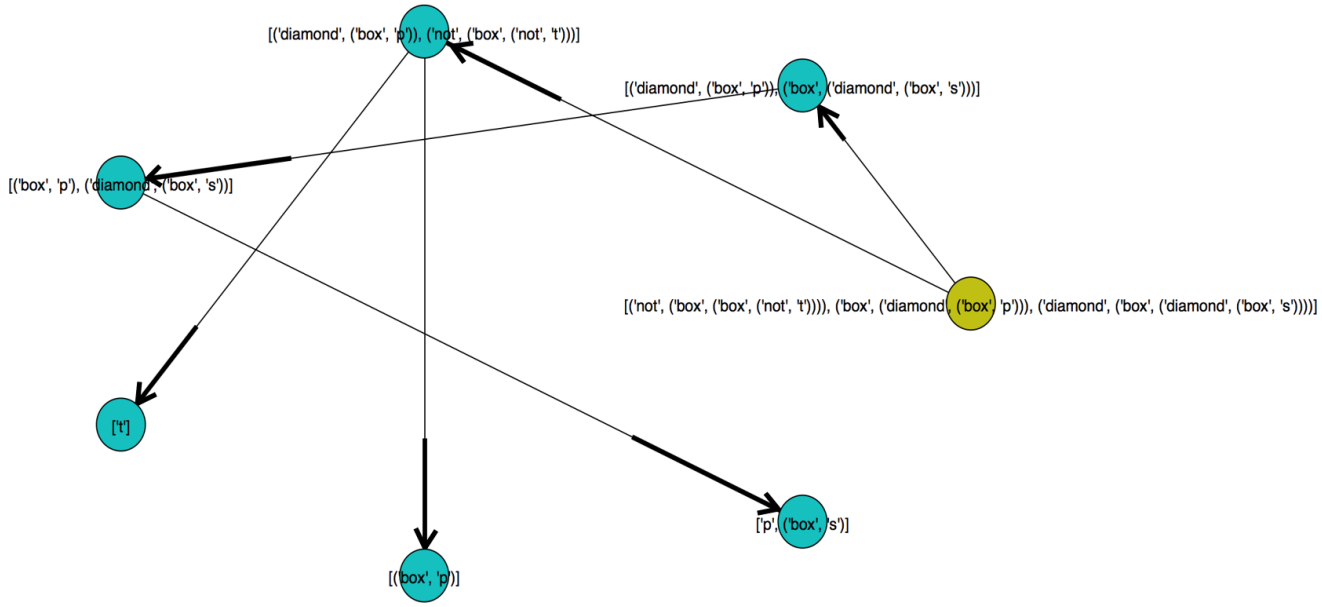
Please note that for each implemented algorithm we start by creating a graph with a single node that includes the input formula indicated by a yellow node. This indicates that it is in fact the initial node at which we started expanding our formula and in consequence our model.

4.2. **Logic K.**

We start with the simplest modal logic K which does not have any restrictions on the Kripke frame. The following examples present what our algorithms produce as an output.

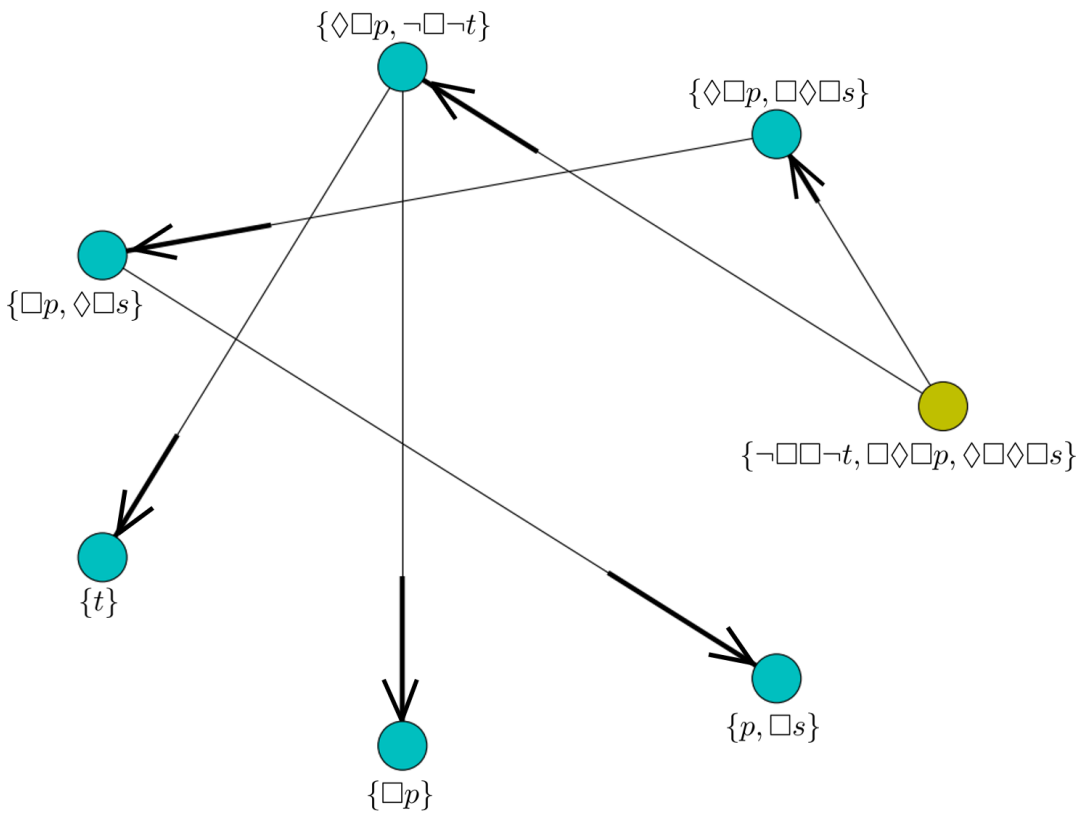
Example 4.1. *Consider input formula* $= ((\Diamond\Box\Diamond\Box s \wedge \Box\Diamond\Box p) \wedge \neg\Box\Box\neg t)$.

Only in this example we do show here a graph that our algorithm returns as other cases would not be easily readable on the page. It is important to acknowledge that the labels used for the node are in prefix format which allows our programs to successfully verify the satisfiability of the formula in the specific axiom. The following graph is the model that we receive directly as an output from the program:



In order to improve the presentation and the readability of the report, we converted all labels into a format that can be much easily interpreted. Such a modification allowed us to fit in larger graphs because the initial prefix format can produce very lengthy formulas. The following graph is exactly the same as the one above, except that it contains modified labels. We will be using the same modified format in all examples, presented in this section.

Please note that the algorithms are fully automated and they cannot be modified in any way, therefore the only modification that we made after our program produced the results were labels. With the modified labels for this examples we get the following result:



The algorithm outputs one graph in which the input formula is satisfiable. Consequently, this is a model for $((\Diamond\Box\Diamond\Box s \wedge \Box\Diamond\Box p) \wedge \neg\Box\Box\neg t)$ *in modal logic K.*

Example 4.2. *Let's consider a valid formula* $= ((\Box p \wedge \Box(p \to q)) \to \Box q)$
In order to verify that this formula is valid we need to check whether its negation is unsatisfiable, so the input formula that we use is $\neg((\Box p \wedge \Box(p \to q)) \to \Box q)$.

The negated formula is placed on the single node in the initial graph. Our implementation allows us to see the initial step and the graph with one node which is as follows:

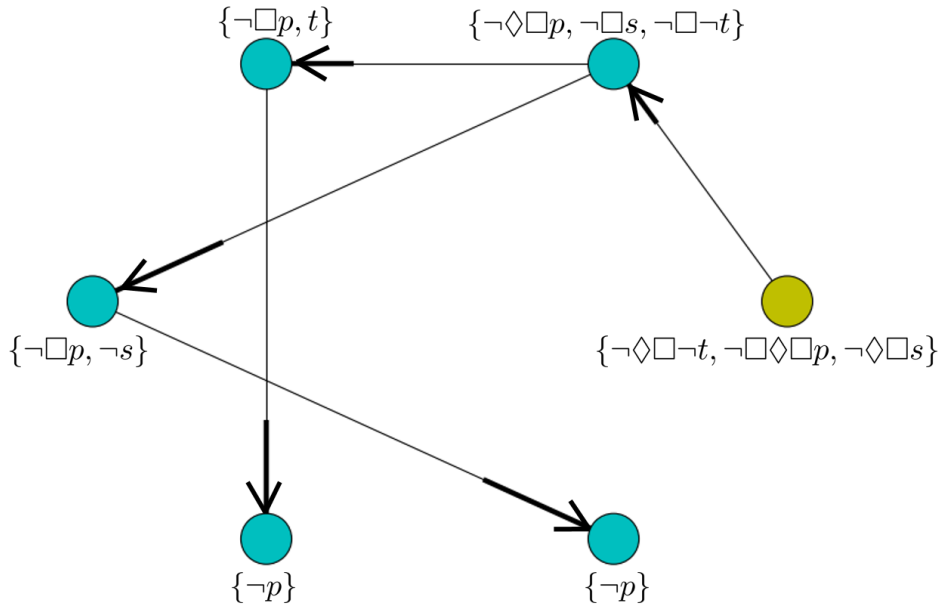
$$\{\Box p, \Box(p \to q), \neg\Box q\}$$

Our program does not return any models since the negated formula is unsatisfiable. Hence, the picture above only illustrates the initial step when the formula is placed in the single node. For this particular example we also include a message from the terminal which we will see when our formula is unsatisfiable. The following is the exact representation of the output that we receive:

```
There are no models for the input formula:  ~(([]p ^ [](p -> q)) -> []q)
So the the negation of it :  ~( ~(([]p ^ [](p -> q)) -> []q) ) is valid.
```

The message from the terminal again does not use the diamond and box symbols that are available to us in Tex. However, we can even so easily verify what formula is unsatisfiable and as a consequence what formula is valid in axiom K. In this case the second line of the output we can see that there is double negation at the start of the formula, therefore what we really get is $((\Box p \wedge \Box(p \to q)) \to \Box q)$, *which is of course what we expected.*

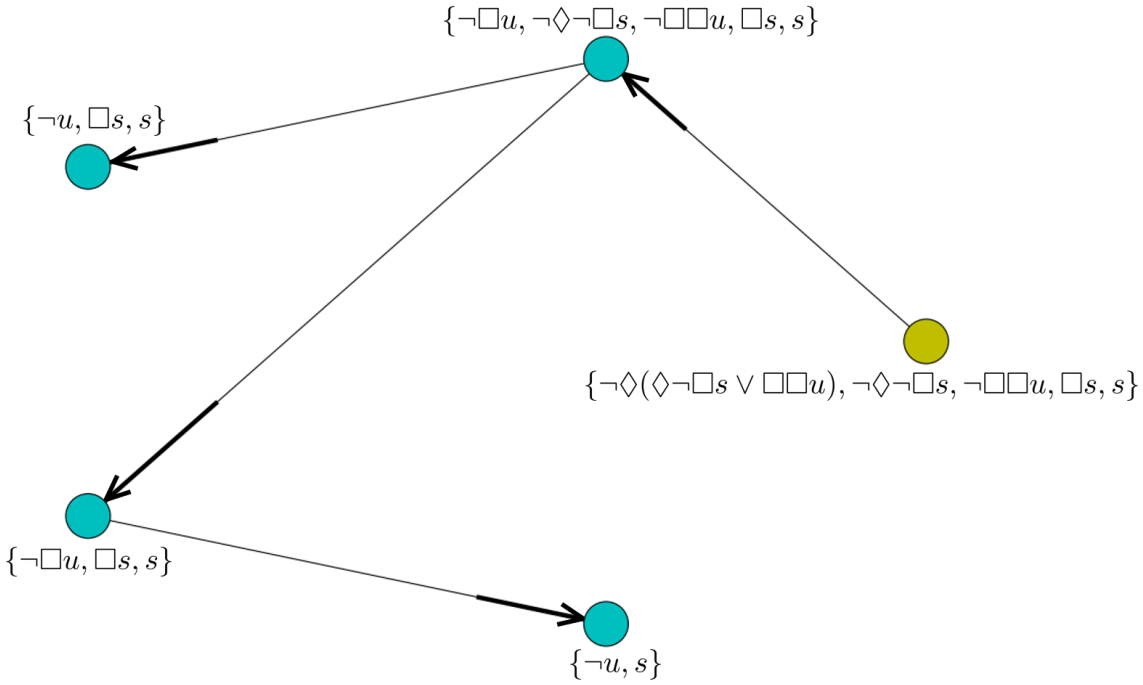
Example 4.3. *Consider input formula* $= (\neg(\neg\Diamond\Box s \to \Box\Diamond\Box p) \wedge \neg\Diamond\Box\neg t)$.
The output of the algorithm gives us the following graph:



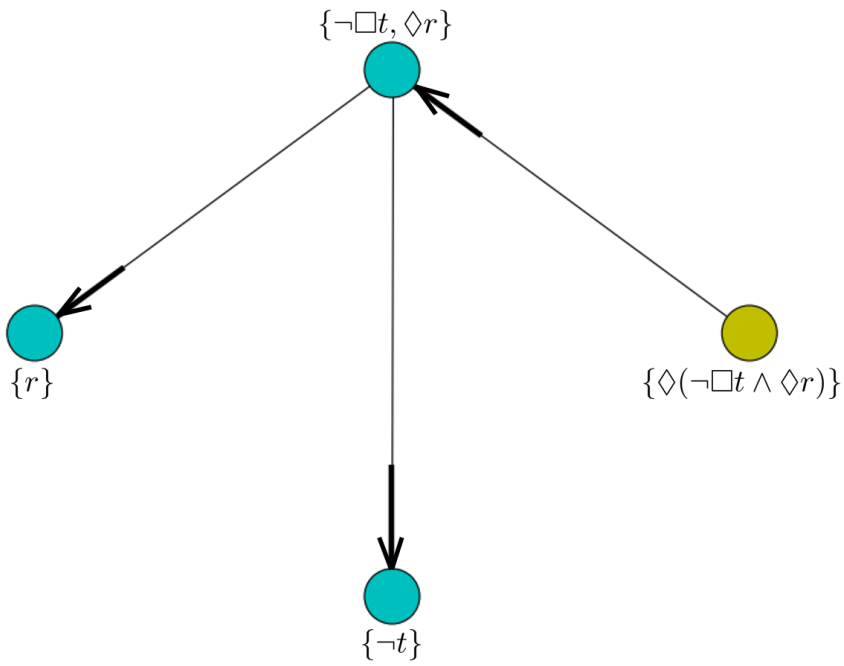
For our input formula we have one model in which the formula is true, meaning that our formula is satisfiable.

4.3. **Logic T.**
T axiom is the class of all reflexive frames so all models presented in this section satisfy this property. Graphs do not include self loops for visual purposes but every node and set of formulas corresponding to it satisfies reflexivity.

Example 4.4. *Consider input formula* $= (\Diamond(\Diamond\neg\Box s \vee \Box\Box u) \to \Diamond(\neg\Box t \wedge \Diamond r))$.
In this case there is more than one possible model for which the formula is true. The output of the algorithm gives us the following graphs:
Model 1:

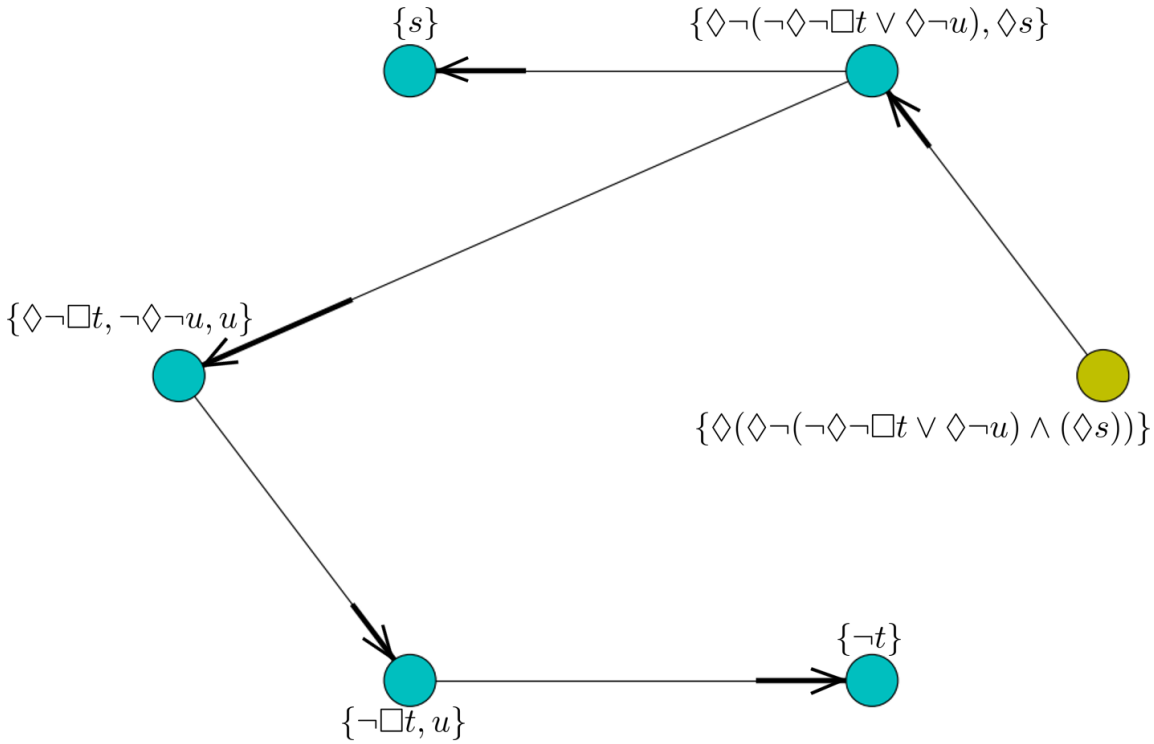


Model 2:



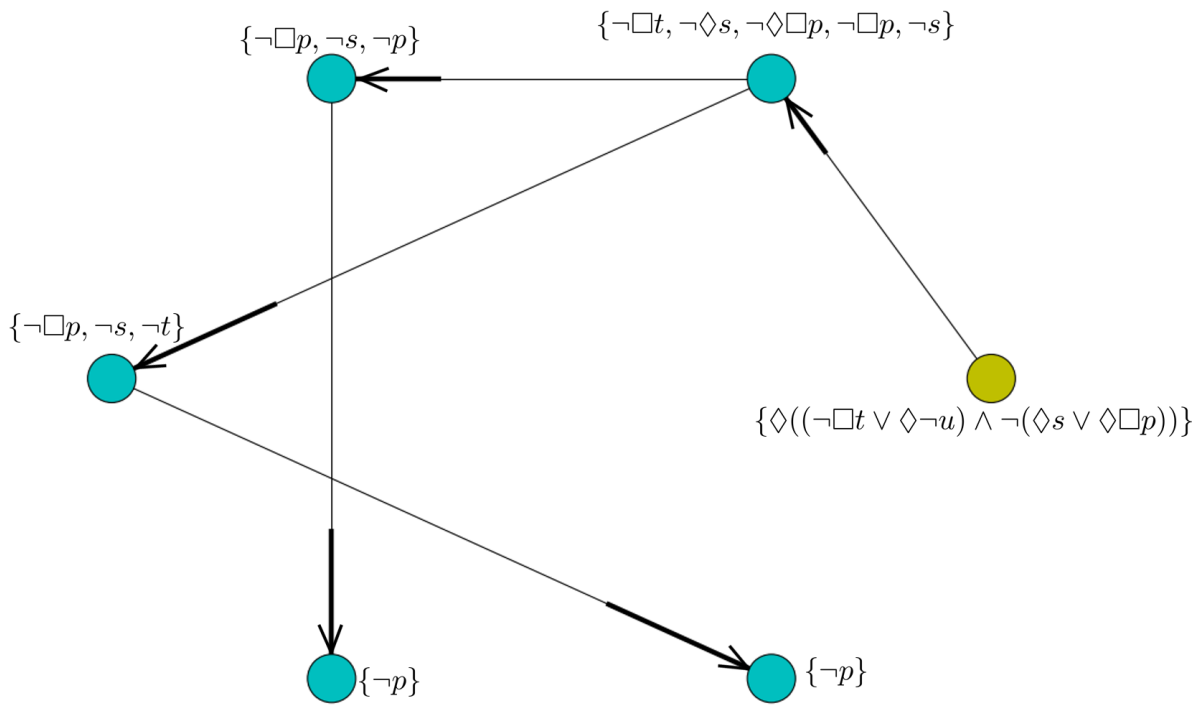
The reason the algorithm produced two models is that we have beta formulas as part of our input formula. Clearly we have a satisfiable formula in the class of all reflexive frames.

Example 4.5. *Consider input formula* $= (\Diamond(\Diamond\neg(\neg\Diamond\neg\Box t \vee \Diamond\neg u) \wedge (\Diamond s)))$
The output of the algorithm gives us the following graph:

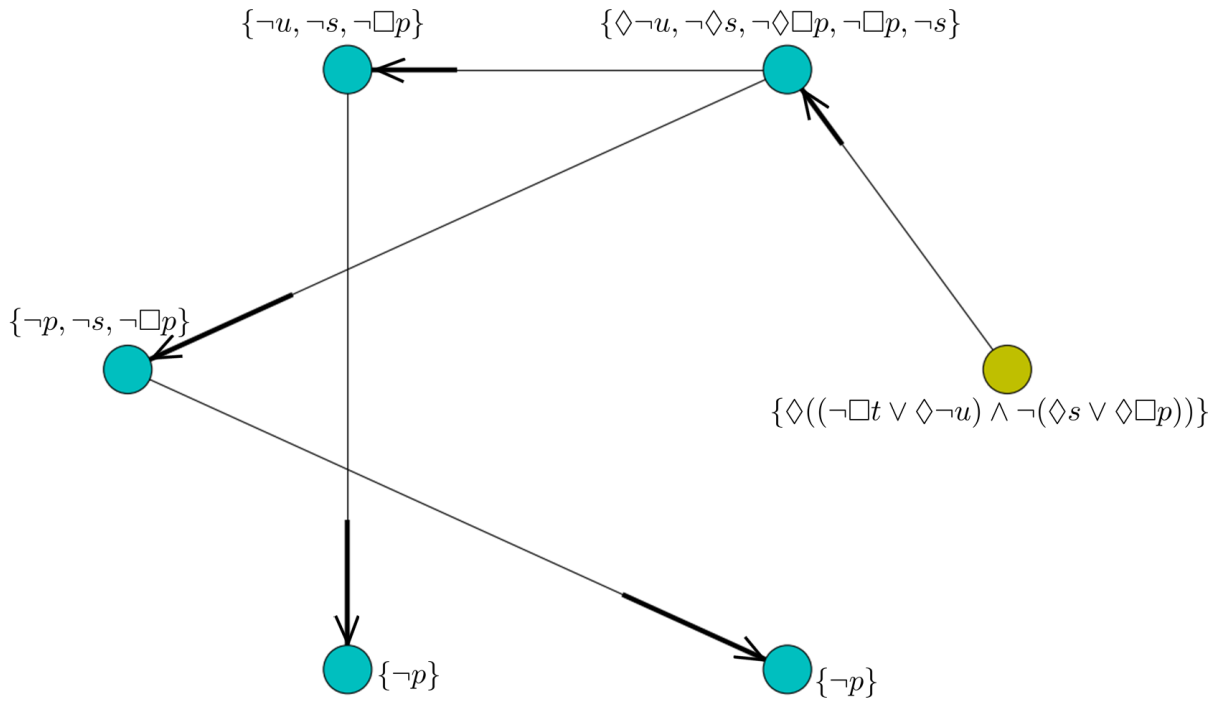


Here the algorithm outputs only one satisfiable model. Again, the model is satisfying the reflexive relation and the input formula is true in this model.

Example 4.6. *Consider input formula* $= (\Diamond((\neg\Box t \vee \Diamond\neg u) \wedge \neg(\Diamond s \vee \Diamond\Box p)))$
The output of the algorithm gives us the following graphs:
Model 1:

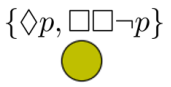


Model 2:



In this instance also we get two models in which our formula is true. The input formula is satisfiable in the class of all reflexive frames.

Example 4.7. *Consider input formula* $= (\Diamond p \wedge \Box\Box\neg p)$. *We selected this formula to demonstrate that formulas which are satisfiable in logic K are not necessary satisfiable in logic T. In fact we verified through our T solver that this formula is unsatisfiable, having the following output:*



Our program does not return any models since the formula is unsatisfiable in axiom T. Hence, the picture above only illustrates the initial step when the formula is placed in the single node of the initial graph. For this example we also include a message from the terminal which we will see when our formula is unsatisfiable. The following is the exact representation of the output that we receive:

```
There are no models for the input formula:  <>p ^ [][]~p
So the the negation of it :  ~( <>p ^ [][]~p ) is valid.
```

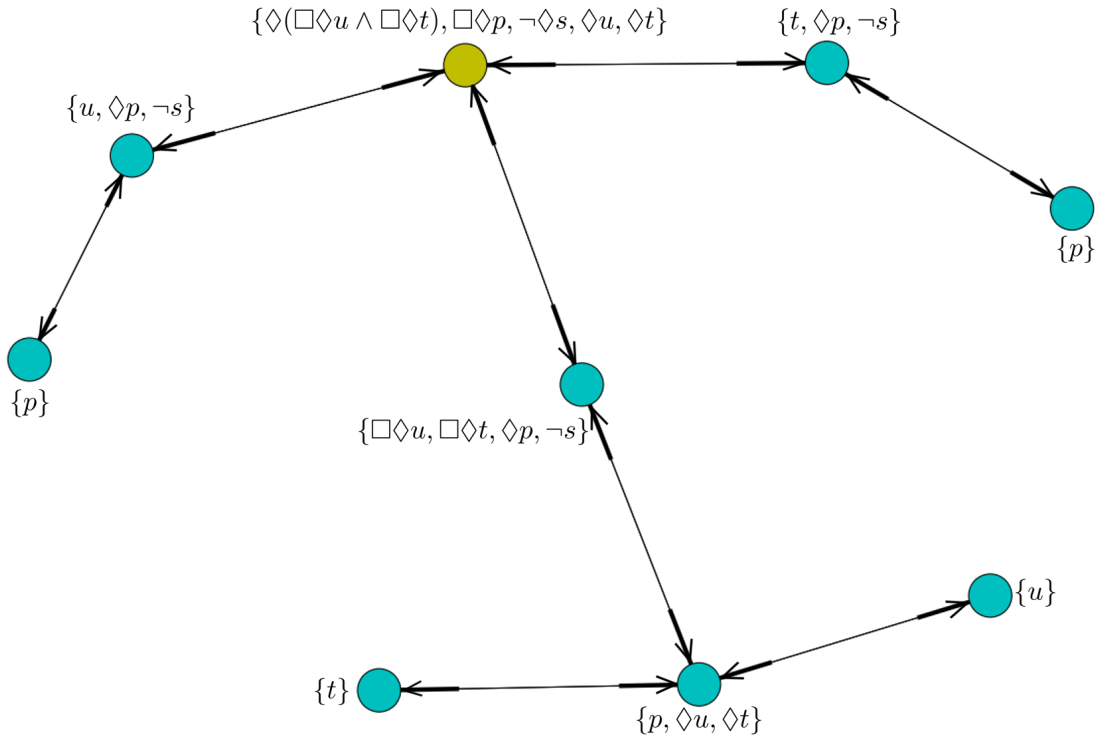
The message from the terminal confirms that the formula $(\Diamond p \wedge \Box\Box\neg p)$ *does not have any models in axiom T, therefore, it is unsatisfiable. It is a formula which is unsatisfiable in the class of reflexive frames, however it is satisfiable in the axiom system K. For confirmation of this, the following is the model for the input formula returned by the K solver:*

Since we have a model we know that the input formula is satisfiable in logic K. This is an illustration of the fact that formulas satisfiable in logics such as K are not necessarily true in stronger logics.

4.4. **Logic KB.**

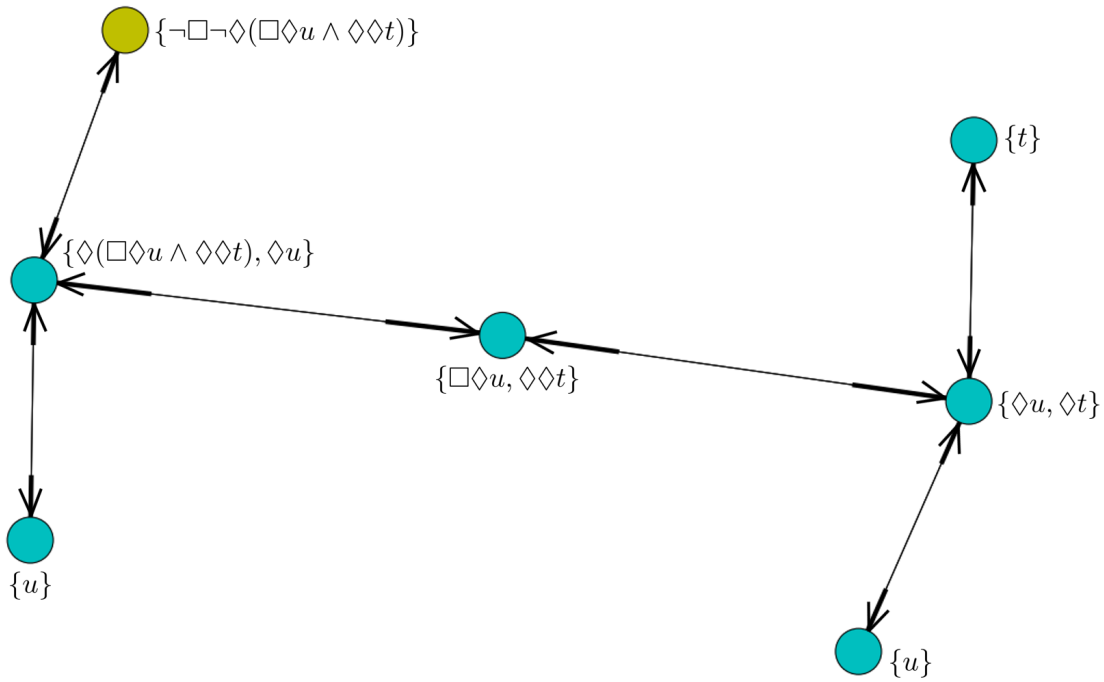
KB axiom is the class of all symmetric frames. So, each pair of connected nodes has a bi-directional arrow and satisfies the symmetric property.

Example 4.8. *Consider input formula* $= (\neg(\Diamond(\Box\Diamond u \wedge \Box\Diamond t) \rightarrow (\Box\Diamond p \rightarrow \Diamond s)))$.
The output of the algorithm gives us the following graph:

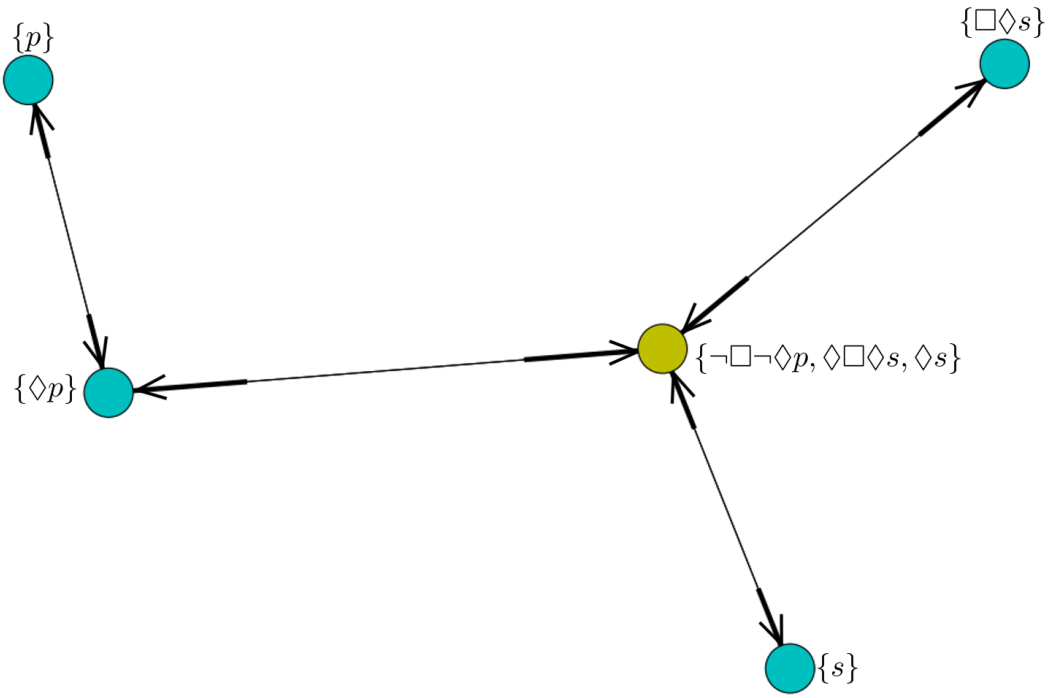


The KB axiom solver returned one model in which the input formula is true. The model satisfies the symmetric relation and therefore our formula is satisfiable in the class of symmetric frames.

Example 4.9. *Consider input formula* $= (\Box\neg\Diamond(\Box\Diamond u \wedge \Diamond\Diamond t) \rightarrow (\neg\Box\neg\Diamond p \wedge \Diamond\Box\Diamond s)))$.
The output of the algorithm gives us the following graphs:
Model 1:



Model 2:



Again we can observe that for the input formula we get two models as an output in which the formula is true. It is again caused by the beta formulas which are part of the input formula.

Example 4.10. *Consider input formula* $= ((\neg\Box\neg\Diamond(\Box\Diamond u \wedge \Diamond\Diamond t) \wedge (\neg\Box\neg\Diamond p \wedge \Diamond\Box\Diamond s)) \wedge \Box\neg s)$.
For this formula we did not assume that it could be unsatisfiable, inputting its negation. However, it turns out that the input formula is in fact unsatisfiable and therefore we can only present the initial step where the graph only contains a single node with the formula in it:

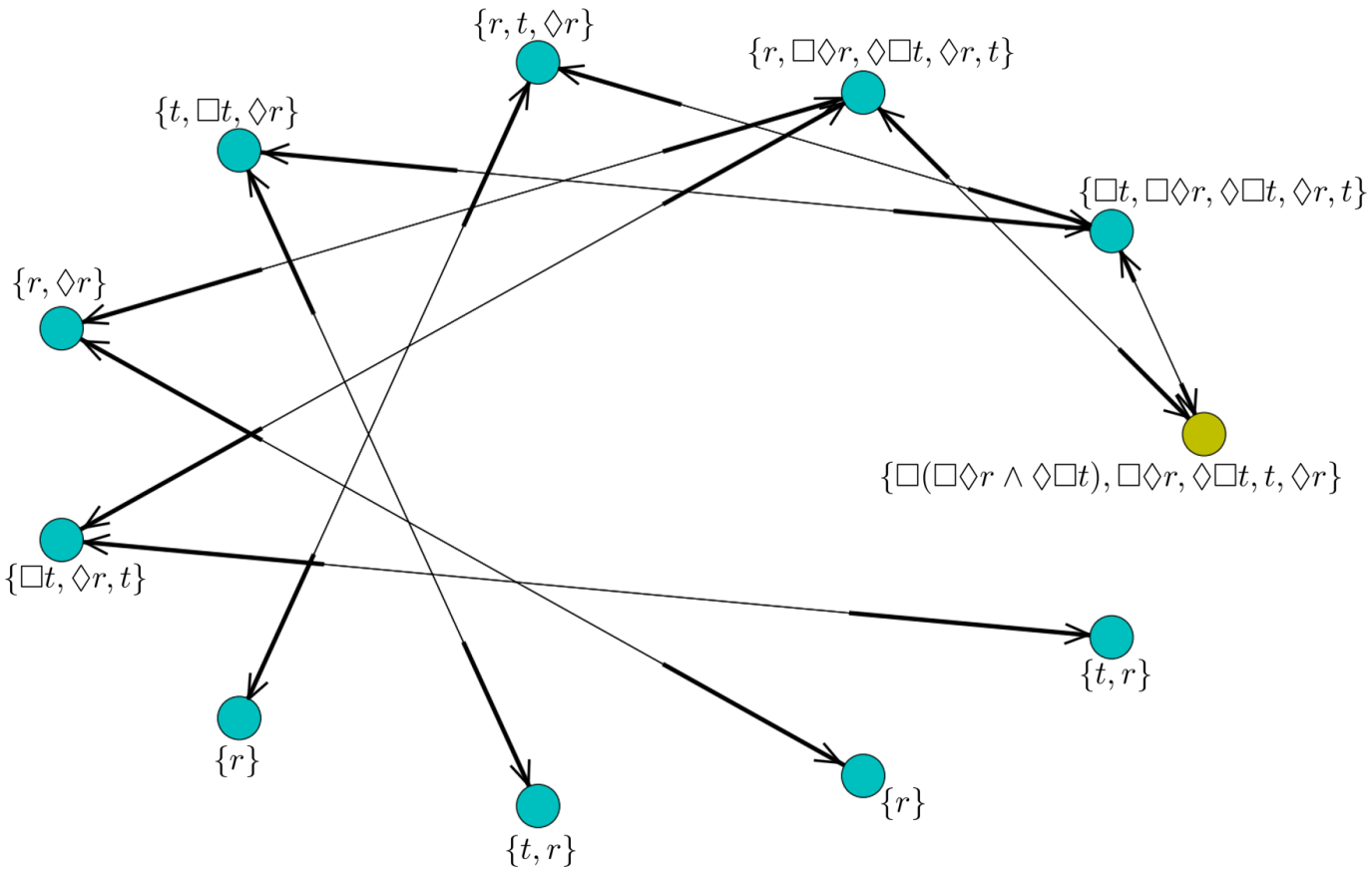
$$\{\neg\Box\neg\Diamond(\Box\Diamond u \wedge \Diamond\Diamond t), \neg\Box\neg\Diamond p, \Diamond\Box\Diamond s, \Box\neg s\}$$

The algorithm does not return any models. Consequently, this tells us that the negation of it is valid. Therefore, $\neg((\neg\Box\neg\Diamond(\Box\Diamond u \wedge \Diamond\Diamond t) \wedge (\neg\Box\neg\Diamond p \wedge \Diamond\Box\Diamond s)) \wedge \Box\neg s)$ *is a valid formula in the class of all symmetric frames. The following is the exact representation of the output that we receive in that case:*

```
There are no models for the input formula:  (~[]~<>([]<>u ^ <><>t) ^ (~[]~<>p ^ <>[]<>s)) ^ []~s
So the the negation of it :  ~( (~[]~<>([]<>u ^ <><>t) ^ (~[]~<>p ^ <>[]<>s)) ^ []~s ) is valid.
```

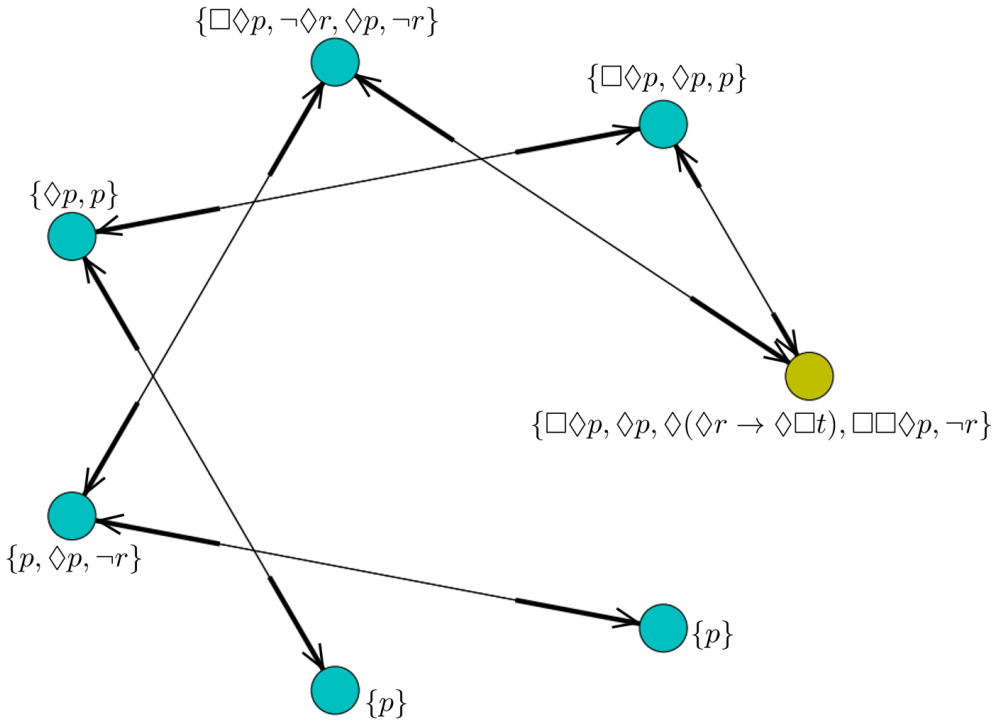
4.5. **Logic B.**
B axiom is the class of symmetric and reflexive frames. Again, we do not include self loops in any of the graphs, for readability purposes, however all graphs satisfy reflexive and symmetric relations. Similarly to the previous section, our graphs include bi-directional arrows between each pair of connected nodes.

Example 4.11. *Consider input formula* $= (\Box(\Box\Diamond r \wedge \Diamond\Box t))$.
The output of the algorithm gives us the following graph:

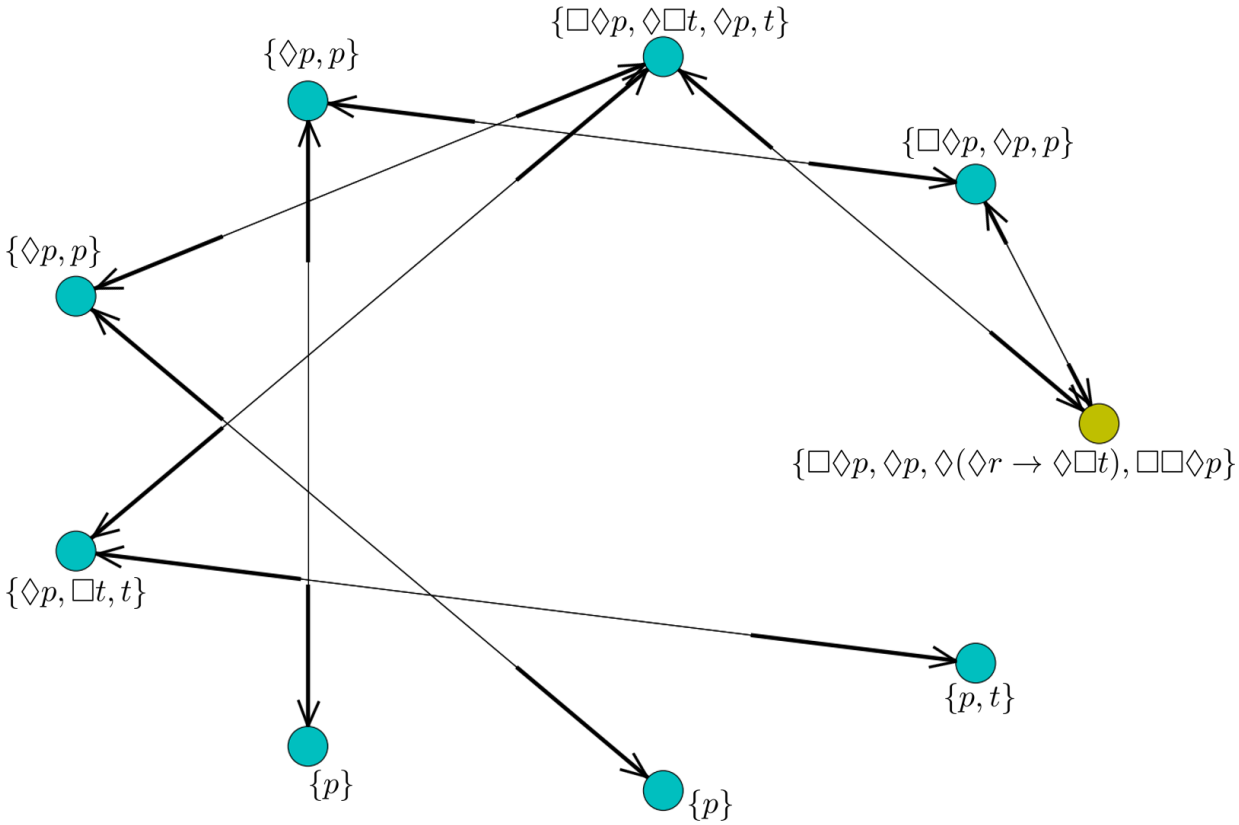


The B axiom solver returned one model in which the input formula is true. Therefore, the formula is satisfiable in the class of symmetric and reflexive frames.

Example 4.12. *Consider input formula* $= ((\Diamond(\Diamond r \rightarrow \Diamond\Box t)) \wedge (\Box\Box\Diamond p))$.
The output of the algorithm gives us the following graphs:
Model 1:

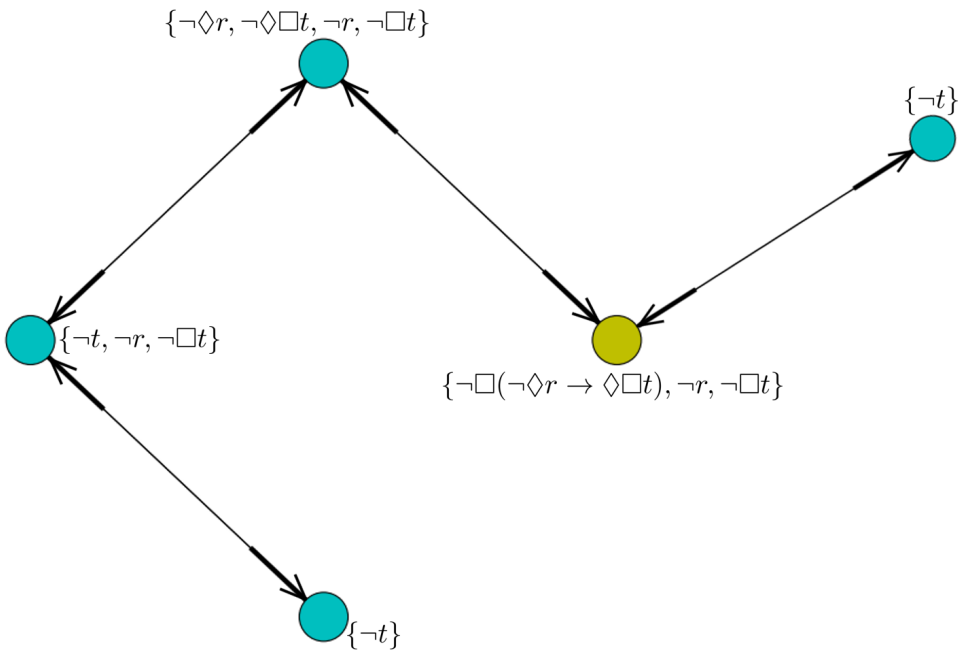


Model 2:

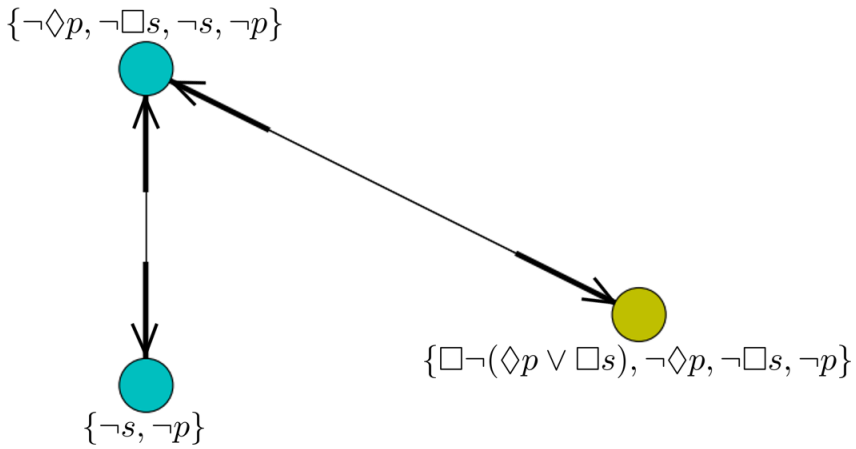


We have two models in which our formula is true, again because we have a beta formula as part of the input formula.

Example 4.13. *Consider input formula* $= (\neg(\Box(\neg\Diamond r \rightarrow \Diamond\Box t) \wedge \neg\Box\neg(\Diamond p \vee \Box s)))$
The output of the algorithm gives us the following graphs:
Model 1:



Model 2:



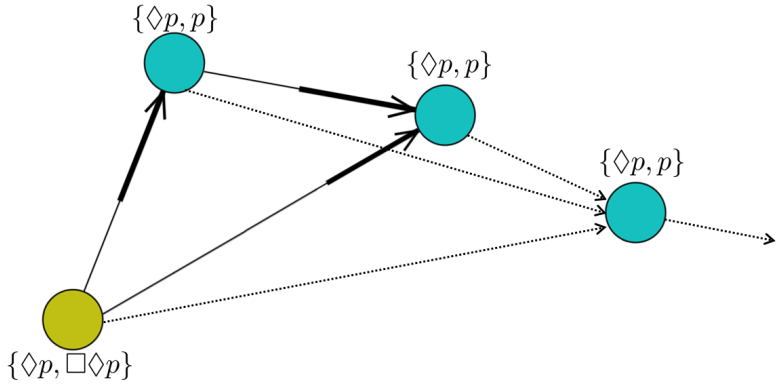
This is another formula in which we get more than one model that makes the input formula satisfiable in the class of symmetric and reflexive frames.

4.6. **Logic K4.**

Now we reach the section in which we consider transitive frames. K4 and S4 axioms satisfy the transitivity relation which makes them more difficult problems to solve. In previous sections we have explained that their complexity is PSPACE-Complete; they are much more difficult to solve in comparison with other axioms presented in this report. In practice this means that without additional checking processes for some input formulas our algorithms (K4 and S4) will not terminate and therefore they will not output a finite model.

To ensure that K4 and S4 solvers terminate and produce finite models we have implemented a loop checker which we presented in the previous section as part of algorithms 6b, 7b. The job of the loop checker is to check whenever there is a new node to be added whether a node with the same formulas already exists. If there is a node in the graph with exactly the same set of formulas, we do not create a new node but we add an edge between the node we expanding and the existing node in the graph. In order to clearly present the importance of the loop checker we will illustrate it on the following example:

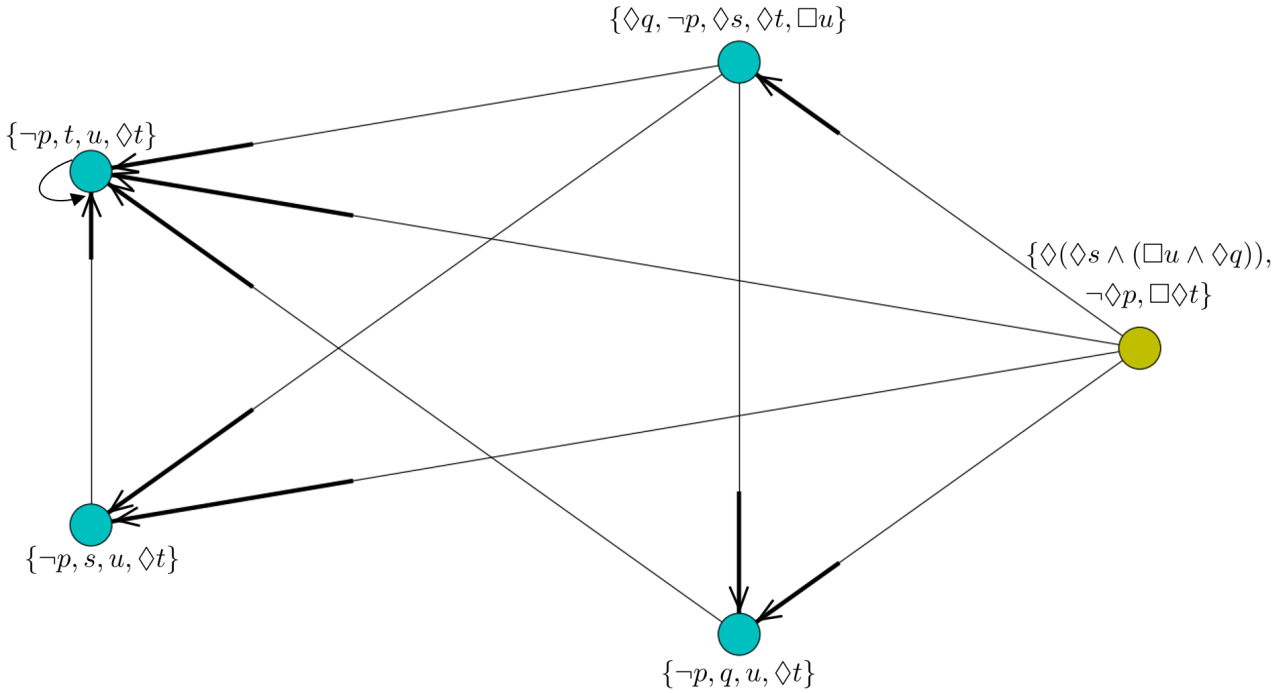
Example 4.14. *Consider input formula* $= \Diamond p \wedge \Box\Diamond p$. *This simple formula does not terminate in the class of transitive frames. This means that although it is satisfiable, our algorithms will not return any models since they will loop to infinity. The following is a model which illustrates the situation where K4 and S4 algorithms do not use loop checking; in consequence the model will be expanding forever without terminating:*



The dotted lines indicate relations that are added when a new node is created. Because of the transitive condition we have to consider all box (gamma) formulas from previous nodes and add their expansions to the new node. We can see that the node with the set $\{\Diamond p, p\}$ *will always create the same node over and over again when we expand it. Therefore, in this instance our program will run forever and a finite model will not be returned. As discussed, for this reason, K4 and S4 solvers implement a loop checking procedure which guarantees termination and results in returning a finite model if one exists. For the input formula here we get the following finite model:*

All the following examples presented in the sections regarding results of K4 and S4 algorithms use this loop checker.

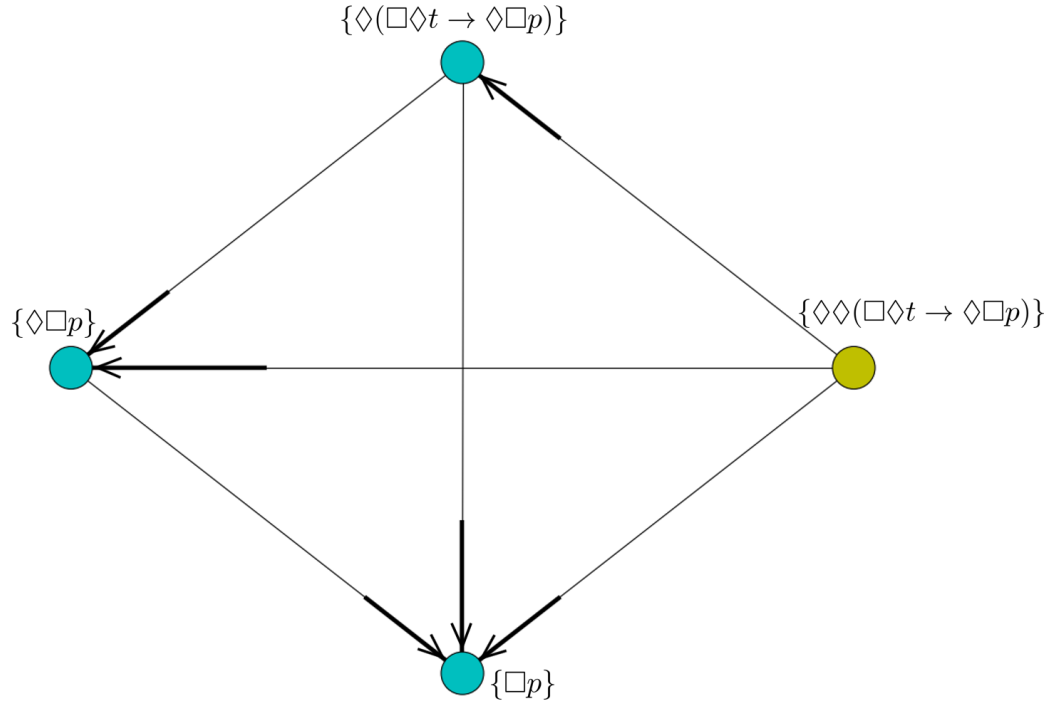
Example 4.15. *Consider input formula* $= (\neg(\neg\Diamond(\Diamond s \wedge (\Box u \wedge \Diamond q)) \vee (\Box\Diamond t \rightarrow \Diamond p)))$.
The output of the algorithm gives us the following graph:



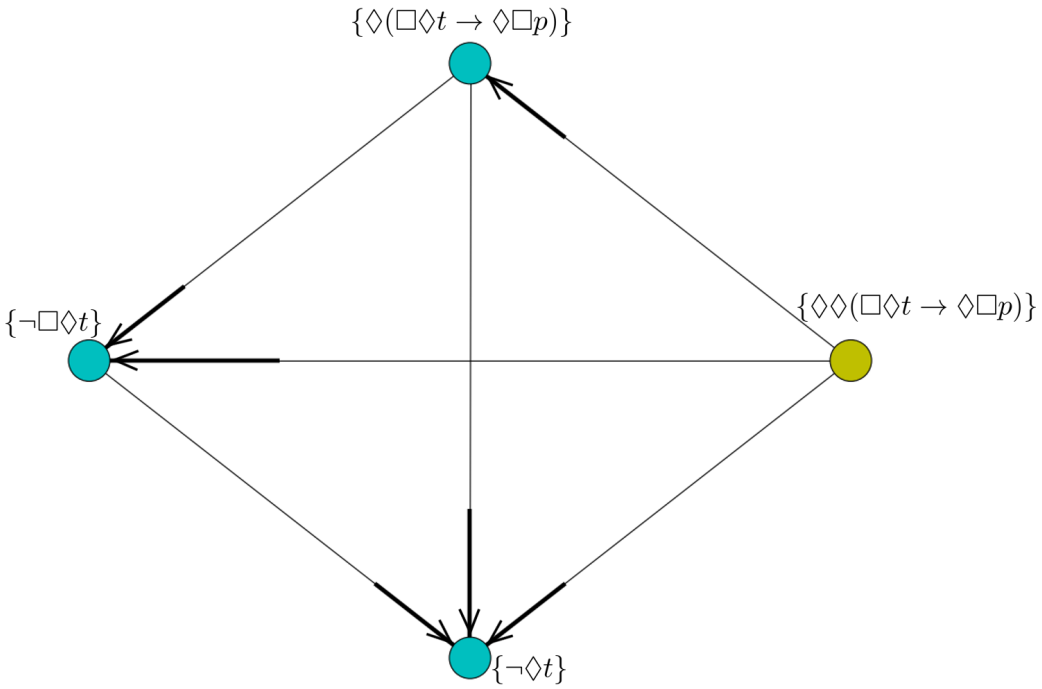
We have one model in which our formula is true. Straight away we can see that the loop checker has been used in order to terminate the program and return the finite model that makes the input formula satisfiable in the

class of transitive frames. Let's look closely at the graph, and specifically at a node containing the following set of formulas: $\{\neg p, t, u, \Diamond t\}$. It is important to mention that K4 is not the class of transitive and reflexive frames, and yet we here have a node with a self loop. This is purely the result of running the loop checker which detected that the same node will be created over and over again. Therefore as a result we have a node with a reflexive relation. Furthermore, we can easily check that if we expand formula $\Diamond t$ from that node, we are going to get the same values, meaning that our loop checker worked well and allowed our program to terminate with a finite model.

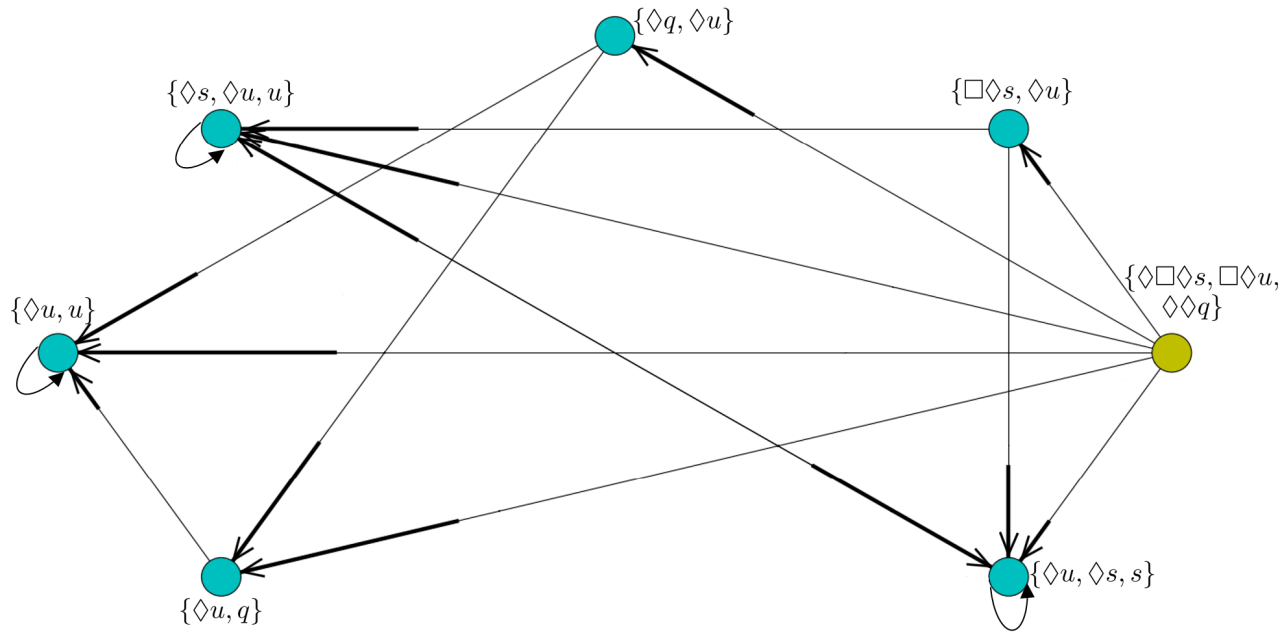
Example 4.16. *Consider input formula* $= ((\Diamond\Box\Diamond s \wedge (\Box\Diamond u \wedge \Diamond\Diamond q)) \vee \Diamond\Diamond(\Box\Diamond t \rightarrow \Diamond\Box p))$.
The output of the algorithm gives us the following graphs:
Model 1:



Model 2:

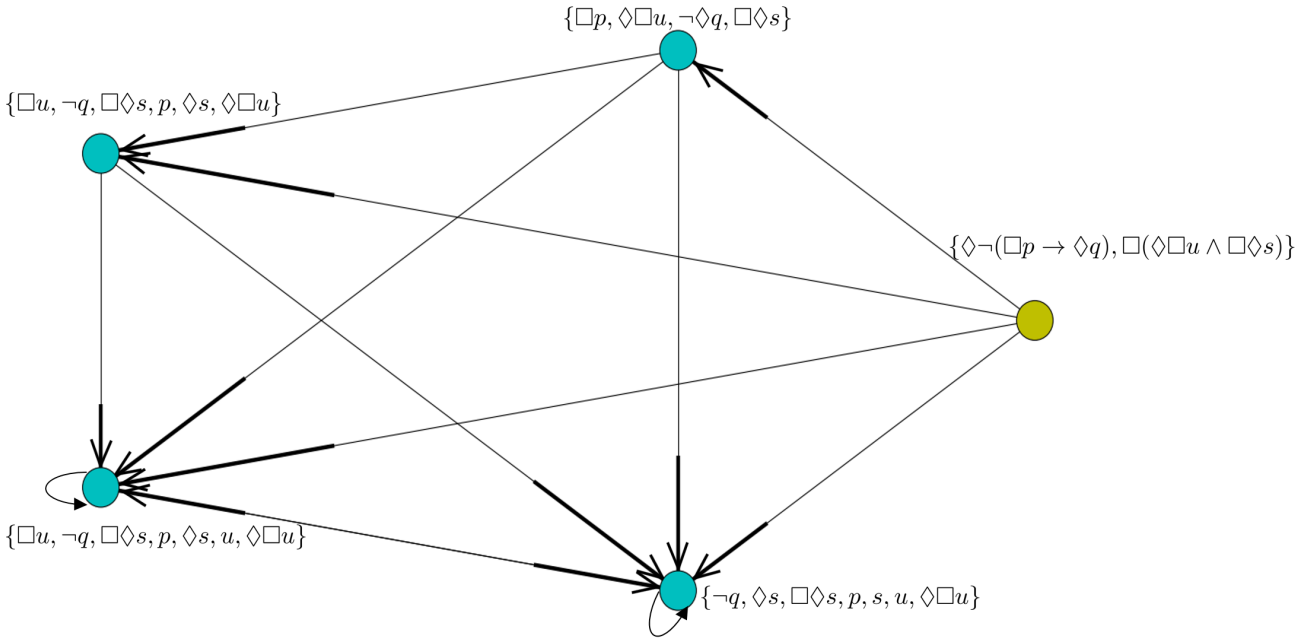


Model 3:



In this example we got three models in which our formula is true. These different models are the consequences of the beta formulas which we expanded during the execution of the algorithm. The first two models are the simple ones, meaning they only include transitive relations as intended by the K4 solver. The third model is more involved as the loop checker has been used to produce a finite model. In that model we have two nodes which are reflexive and additionally they are symmetric. This means that the set of formulas at the those nodes contain two different delta formulas, e.g. $\{\Diamond s, \Diamond u, u\}$*. In such a situation, for example, the first delta formula can be looping to itself and the other delta formula going to a different node which has exactly the same situation. In conclusion, finite models were returned in this instance by the K4 solver and the loop checker again worked well.*

Example 4.17. *Consider input formula =* $(\Box(\Diamond\Box u \wedge \Box\Diamond s) \wedge \Diamond\neg(\Box p \rightarrow \Diamond q))$.
The output of the algorithm gives us the following graph:

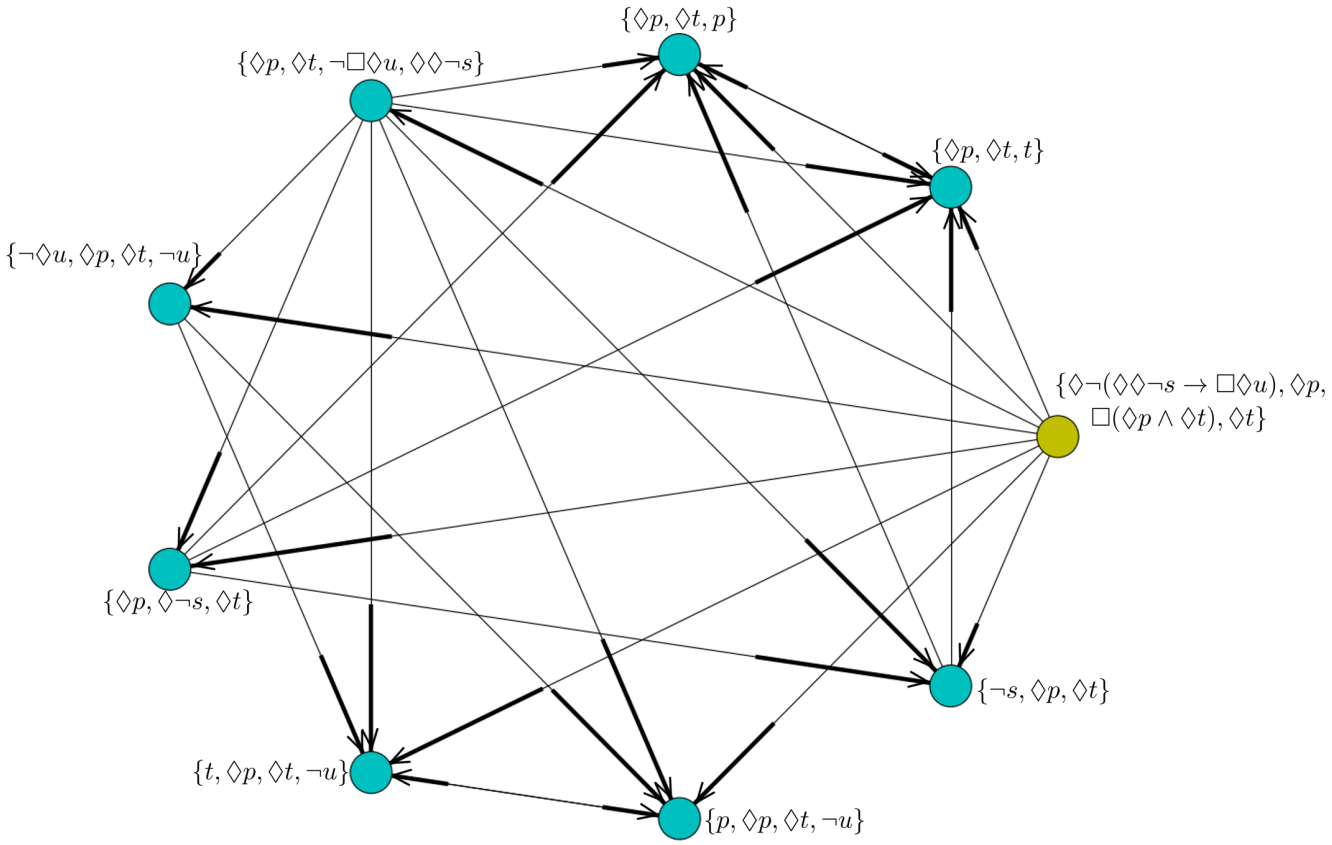


For this input formula the K4 solver returns one model. The model is again more involved and includes two nodes which are additionally symmetric and reflexive. This means that the loop checker has taken action and allowed the solver to terminate with a finite model. The reflexive nodes contain exactly two delta formulas. In each node the first delta formula is the transition to the other node which is part of the symmetric pair, and the second delta formula leads back to the first node which explains the reflexive relation, i.e. self loop.

4.7. **Logic S4.**

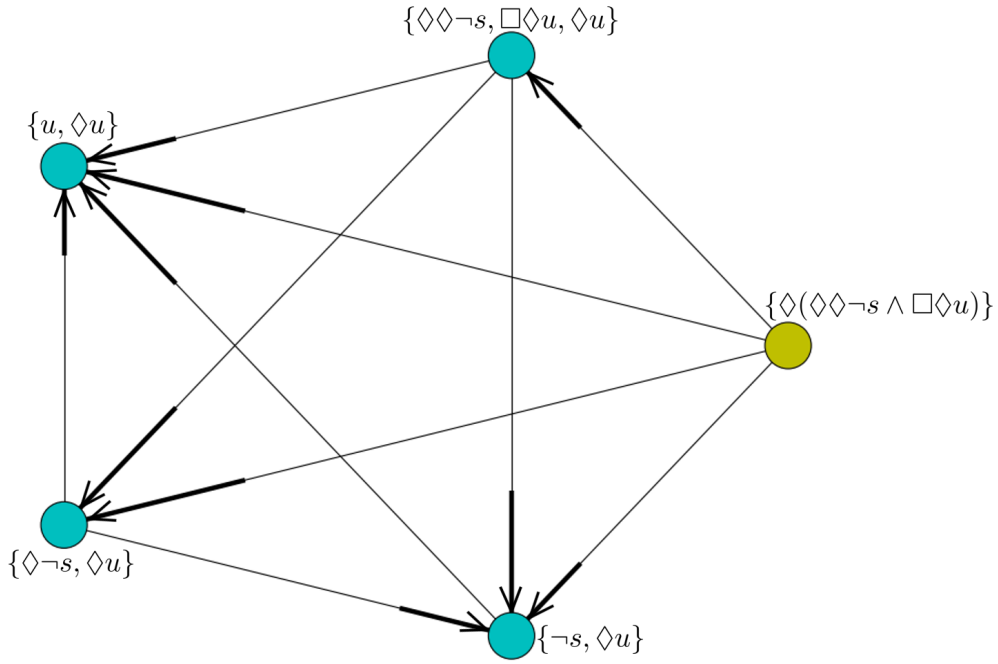
The S4 axiom is the class of transitive and reflexive frames. The S4 solver makes use of the K4 algorithm but in addition uses part of the T algorithm which allows for the model to satisfy the reflexivity relation. Similarly to the examples in previous sections where reflexive models were presented, we do not add self loops to our graphs.

Example 4.18. *Consider input formula* $= (\Diamond\neg(\Diamond\Diamond\neg s \to \Box\Diamond u) \wedge (\Box(\Diamond p \wedge \Diamond t)))$.
The output of the algorithm gives us the following graph:

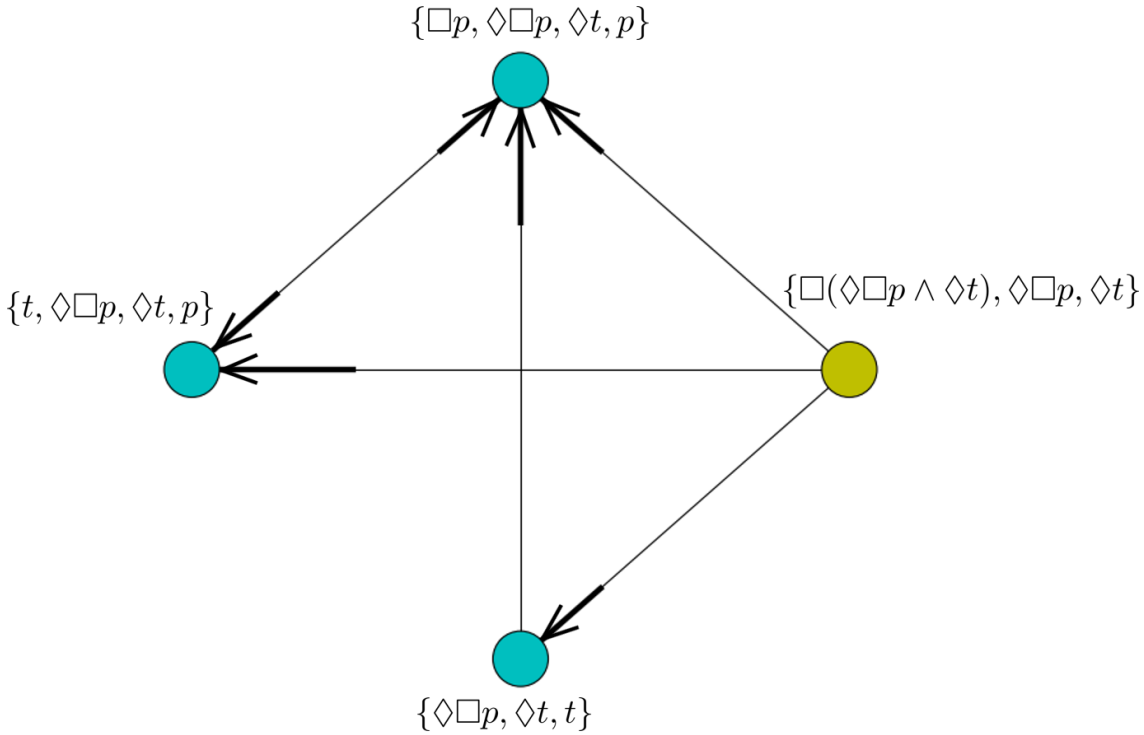


In this example we get one model in which our formula is true. The generated model is more involved and therefore has a more complicated structure. Again, the loop checker has done a great job to actually derive the finite model. We can easily spot that in this example we have two pairs of symmetric nodes. Each pair contains two delta formulas as in the previous example and either remains in the same node or goes to the other node that is in the symmetric relation. Clearly, without the loop checker this model would never terminate; however since a finite model has been delivered that means our input formula is satisfiable.

Example 4.19. *Consider input formula* $= (\Diamond(\Diamond\Diamond\neg s \wedge \Box\Diamond u) \vee (\Box(\Diamond\Box p \wedge \Diamond t)))$.
The output of the algorithm gives us the following graphs:
Model 1:

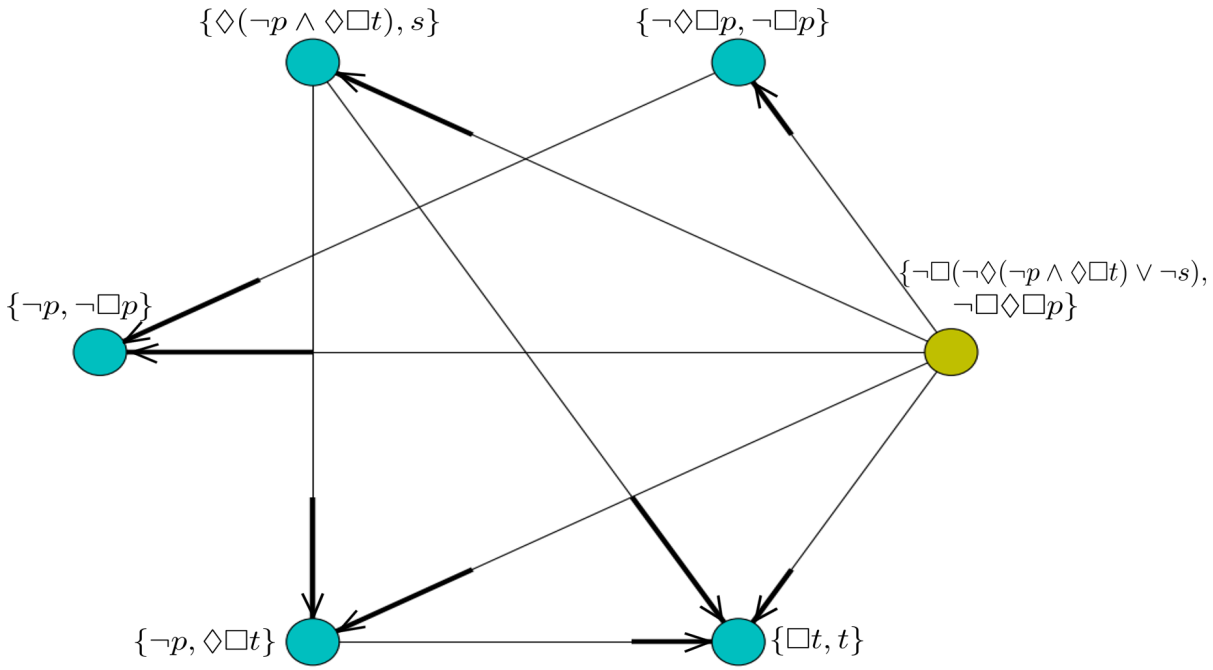


Model 2:

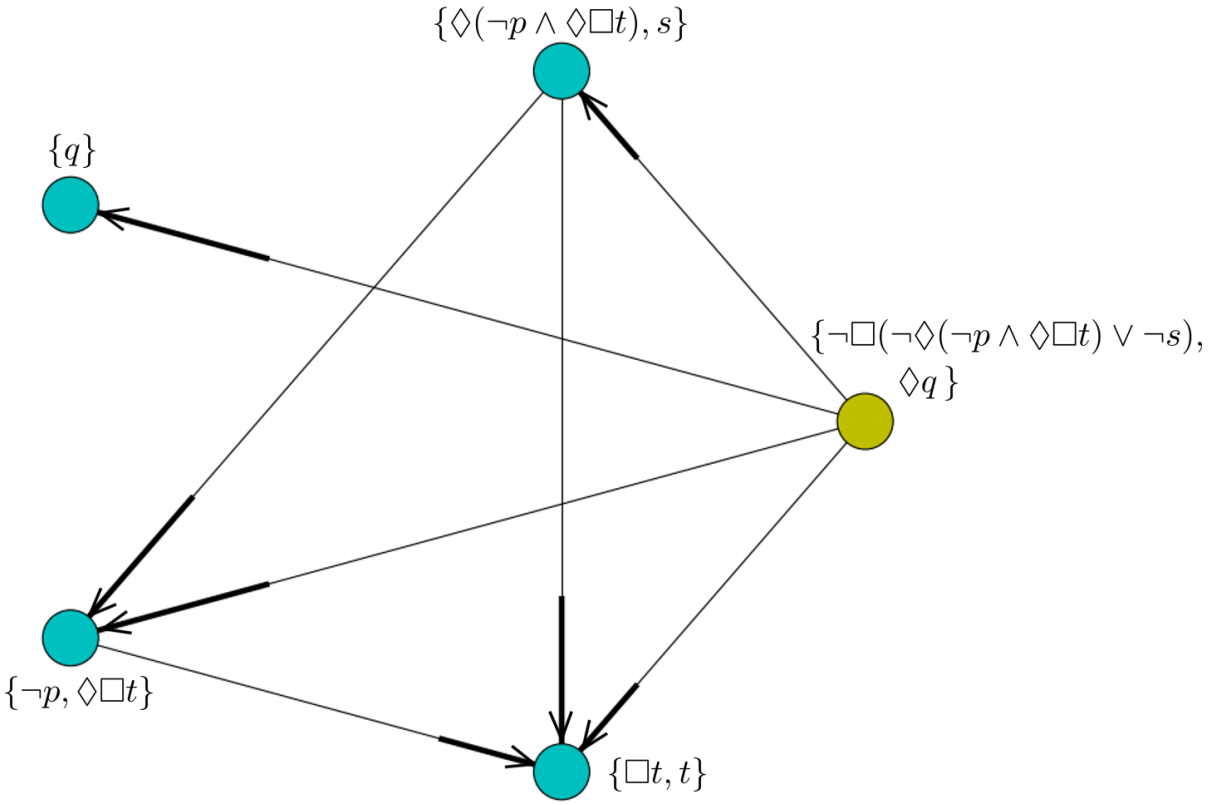


We have two valid models in which the input formula is true. In both models the loop checker has been used to make sure that the S4 solver terminates. In the first model we do not have any symmetric relations; however, we have a node to which all other nodes connect; the value at that node is $\{u, \Diamond u\}$. *In the second model we have a pair of symmetric nodes. This means that if we want to expand one of these nodes, we either reach the same node or we can go to the other node. Thus we know that there are two of the same delta formulas in each of the nodes which justifies the need for a symmetric relationship.*

Example 4.20. *Consider input formula* $= (\neg\Box(\neg\Diamond(\neg p \wedge \Diamond\Box t) \vee \neg s) \wedge (\Box\Diamond\Box p \rightarrow \Diamond q))$.
The output of the algorithm gives us the following graphs:
Model 1:



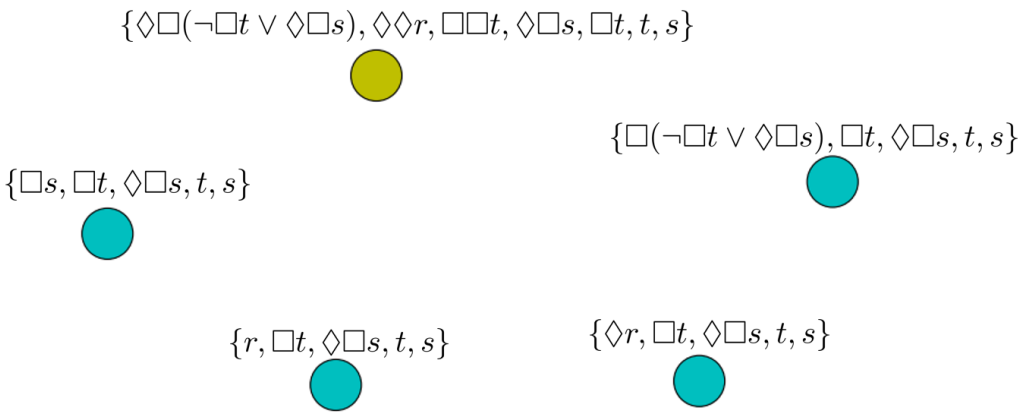
Model 2:



Our final example delivers two models in which the input formula is true. This means that the formula is satisfiable in S4 logic. Again, we note that for both models the loop checker has been used to terminate the process in the S4 solver. In the first model we can either reach the node with $\{\neg p, \neg\Box p\}$ *or* $\{\Box t, t\}$*, whereas in the second model we can either reach* $\{q\}$ *at which point we cannot do anything else, or we can reach* $\{\Box t, t\}$ *which is only expanded by the gamma rule and is also the final node.*

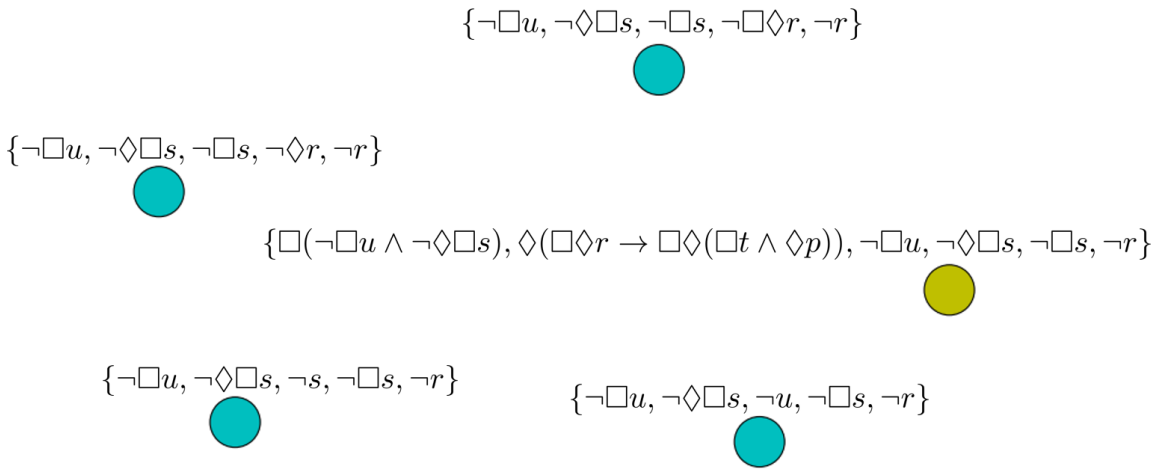
4.8. **Logic S5.**
The S5 axiom is the class of transitive, reflexive and symmetric frames; such a combination is called an equivalence relation. Because of the equivalence relation we were able to maximally simplify our graphs such that generated graphs are simply disconnected nodes. However, we need to remember that each model satisfies transitive, reflexive and symmetric relations. The reason we decided to remove all the edges from graphs in the S5 solver is because of the visual presentation which is greatly improved with this simplified structure. If we were going to generate strongly connected graphs for every lengthy formula, we would quickly find that labels are covered by edges and arrows.

Example 4.21. *Consider input formula* $= ((\Diamond\Box(\neg\Box t \vee \Diamond\Box s)) \wedge (\Diamond\Diamond r \wedge \Box\Box t))$.
The output of the algorithm gives us the following graph:

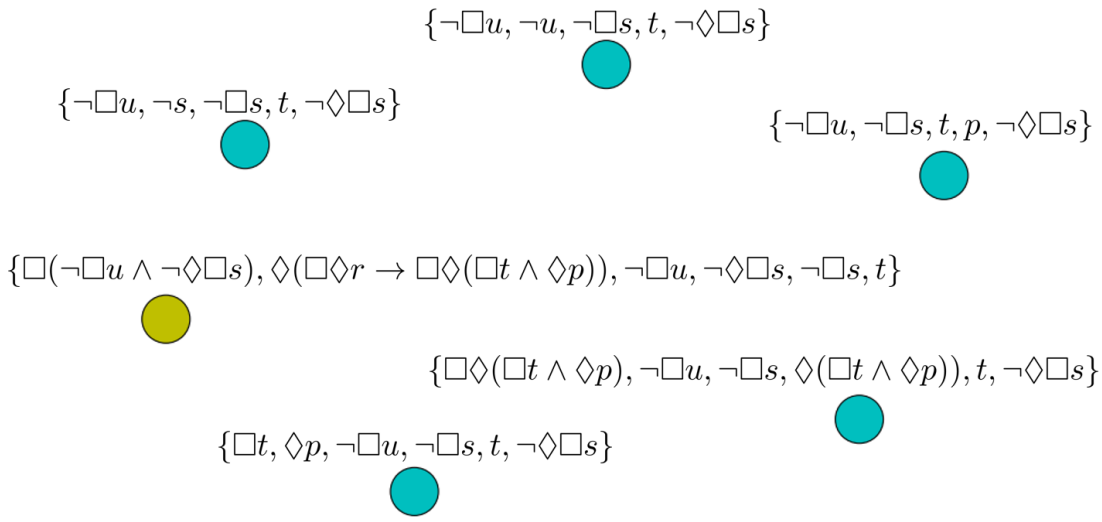


We get one model in which our input formula is true. We can easily verify that the model satisfies an equivalence relation. Therefore, the S5 solver works well with the modifications we applied and we have verified that the formula is satisfiable in S5.

Example 4.22. *Consider input formula* $= (\Box(\neg\Box u \wedge \neg\Diamond\Box s) \wedge \Diamond(\Box\Diamond r \rightarrow \Box\Diamond(\Box t \wedge \Diamond p)))$.
The output of the algorithm gives us the following graphs:
Model 1:

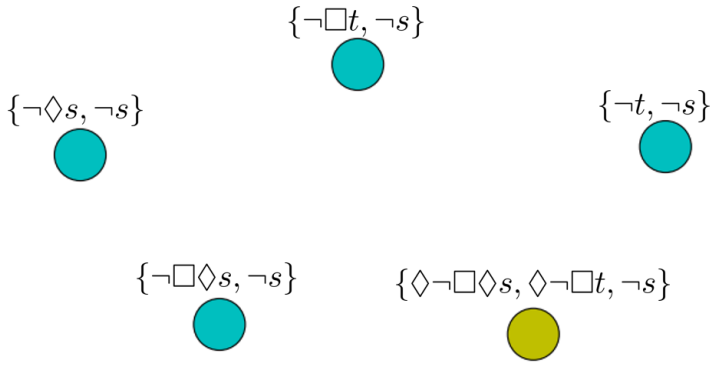


Model 2:

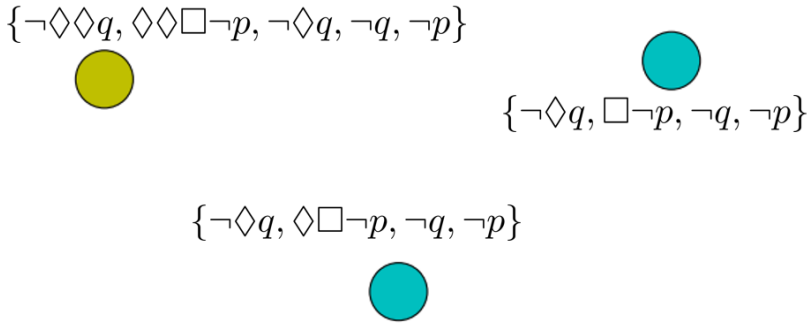


The S5 solver returned two models in which the input formula is true. These models are also simple enough to verify that transitive, reflexive and symmetric relations are satisfied. We expected to have more than one model here since the input formula includes beta formulas. Therefore, the S5 solver verified that the input formula is satisfiable.

Example 4.23. *Consider input formula* $= ((\neg\Diamond\Diamond\Box\neg p \vee (\neg\Box\Diamond p \wedge \Diamond\Diamond q)) \rightarrow (\Diamond\neg\Box\Diamond s \wedge \Diamond\neg\Box t))$.
The output of the algorithm gives us the following graphs:
Model 1:



Model 2:



In our last example the S5 solver again outputs two models in which the input formula is true. Even though the input formula is long we have two simple models which are satisfying an equivalence relation and because of that, the input formula is satisfiable in the logic S5.

4.9. **Testing, Comparison and Verification of Results against Spartacus.**
In order to verify that we get correct results in our programs we used Spartacus software. Spartacus is a tableaux prover for hybrid logic with global modalities. It is one of the solvers available online for free which we can use to challenge our tableau solvers. Spartacus is a very efficient tableau solver so we are expecting it to be faster than our implementations. Furthermore, it is the first system to take advantage of a pattern-based blocking technique that allows it to achieve termination. This technique is a novel blocking technique for converse-free modal and hybrid logics [KS09a][KS09b]. In addition Spartacus implements a number of optimisations, including both established and new techniques [GKS10] [Got09]. It is available at: www.ps.uni-saarland.de/spartacus. However, as will be noted below, Spartacus is not in fact in the domain of application of our theorem provers.

Hybrid logic is an extension of modal logic which additionally includes $\forall$ and $\exists$ quantifiers, and so-called nominals which are propositional symbols of a new sort, each being true at exactly one possible world. In order to compare our tableaux solvers against Spartacus, we can use only K-axiom, reflexive (T-axiom) and transitive (K4-axiom) solvers, those parts of Spartacus which strictly belong to basic modal logic. We can only compare results in axioms K, T and K4. Spartacus software does not produce visual representations of the models, however it returns an answer to whether a formula is satisfiable and also tells us the time it takes to verify that.

In order to compare results with Spartacus we were required to write the input formulas to Spartacus in a different format. The following are examples of modal formulas in the format that Spartacus accepts:

- **Formula:** $\neg(\Diamond p \rightarrow \Box\Box p)$
  (1) **K:** ∼(<r>p -> [r][r]p)
  (2) **T:** $\{reflexive : r\}$ ∼(<r>p -> [r][r]p)
  (3) **K4:** $\{transitive : r\}$ ∼(<r>p -> [r][r]p)
- **Formula:** $\Diamond p \wedge \Box(\neg p \vee q)$
  (1) **K:** (<r>p & [r](∼ p | q)
  (2) **T:** $\{reflexive : r\}$ (<r>p & [r](∼ p | q)
  (3) **K4:** $\{transitive : r\}$ (<r>p & [r](∼ p | q)
- **Formula:** $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q))$
  (1) **K:** ([r](p -> q) -> ([r]p -> [r]q))
  (2) **T:** $\{reflexive : r\}$ ([r](p -> q) -> ([r]p -> [r]q))
  (3) **K4:** $\{transitive : r\}$ ([r](p -> q) -> ([r]p -> [r]q))

4.9.1. *K-axiom comparison.*
Tables 6, 7 and 8 below compare the performance of our theorem provers against Spartacus in the case of K-axiom, T-axiom and K4-axiom instances, respectively.

Table 6 K-axiom comparison of times and outcomes; no restrictions on frames.

	Spartacus		**K-solver**	
formula	**time(sec)**	**outcome**	**time(sec)**	**outcome**
$\Diamond p \wedge \Diamond(\neg p)$	0.054	sat	0.213	sat
$\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$	0.001	sat	0.531	sat
$((\Box p \rightarrow q) \wedge \Diamond\Box(\Box q \rightarrow t)) \wedge (\Diamond\neg p \wedge \Box s)$	0.001	sat	0.354	sat
$\neg(\Diamond\Diamond(\Box p \rightarrow \Diamond q) \rightarrow \Box\Diamond\Box(\Box q \rightarrow \Diamond t))$	0.001	sat	0.387	sat
$\neg\Diamond\Box\Diamond(\Box\Diamond p \rightarrow \Diamond\Box q) \wedge \neg\Box\Diamond\Box(\Box\Diamond\ p \wedge (\Box\Diamond s \wedge \Box\Box t))$	0.002	sat	0.676	sat
$\neg(\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q))$	0.001	unsat	0.00057	unsat
$(\Diamond\Box p \wedge \Box\Diamond s) \wedge \Box(\Box\neg p \wedge q)$	0.001	unsat	0.00029	unsat
$\neg((\Diamond\neg p \vee \Box(\neg\Diamond\Diamond p \rightarrow \Box q)) \vee (\Box q \vee \Diamond\Diamond\neg q))$	0.001	unsat	0.00045	unsat

4.9.2. *T-axiom comparison.*

Table 7 T-axiom comparison of times and outcomes; reflexive frames.

	Spartacus		**T-solver**	
formula	**time(sec)**	**outcome**	**time(sec)**	**outcome**
$\Diamond p \wedge \Box(\neg p \vee q)$	0.001	sat	0.366	sat
$\Box\Box\Box p \rightarrow \neg\Box\Diamond\Box(p \wedge \Diamond s)$	0.001	sat	0.948	sat
$\neg\Box\Box\Box(\Diamond p \wedge \Box q) \wedge \neg\Diamond\Box(\Diamond p \rightarrow \Diamond s)$	0.002	sat	0.470	sat

$\neg\Box\Box\Diamond(\Box\Diamond p \wedge \Diamond\Box q) \wedge \neg\Diamond\Box(\Box\Diamond\neg p \wedge \Box\Diamond s)$	0.002	sat	12.38	sat
$\neg\Diamond\Box\Box\Diamond(\Box\Diamond p \rightarrow \Diamond\Box q) \wedge \neg\Box\Diamond\Box(\Box\Diamond\neg p \wedge (\Box\Diamond s \wedge \Box\Box t))$	0.002	sat	13.889	sat
$\neg(\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q))$	0.001	unsat	0.0015	unsat
$\neg(\Box\Diamond(\Box p \rightarrow \Box q) \rightarrow \Diamond(\Box p \rightarrow \Box q))$	0.001	unsat	0.0076	unsat
$\neg(\Box\Box\Diamond(\Box\Box p \wedge \Box\neg q) \rightarrow \Diamond\Diamond(\Box\Diamond p \rightarrow \Box\neg q))$	0.001	unsat	0.00767	unsat

4.9.3. *K4-axiom comparison.*

Table 8 K4-axiom comparison of times and outcomes; transitive frames.

	Spartacus		**K4-solver**	
formula	**time(sec)**	**outcome**	**time(sec)**	**outcome**
$\Box\Box\Diamond\Box p \wedge \Diamond\Diamond\neg p$	0.001	sat	1.56	sat
$(\Box p \rightarrow p) \wedge (\Box(\Box p \rightarrow p) \wedge \neg p)$	0.002	sat	0.212	sat
$\Box(\Diamond\Box p \wedge \Box\Diamond q) \wedge \Diamond\neg(\Box s \rightarrow \Diamond t)$	0.002	sat	0.240	sat
$(\Box\Box p \rightarrow \Diamond p) \rightarrow (\Box(\Box p \rightarrow p) \rightarrow \Box\Box\neg p)$	0.001	sat	0.513	sat
$\neg((\Box\Box p \rightarrow \Diamond p) \vee (\Box(\Box p \rightarrow p) \wedge \Box\Box\neg p))$	0.001	sat	0.335	sat
$\neg(\Box p \rightarrow \Box\Box p)$	0.001	unsat	0.000317	unsat
$((\Box p \rightarrow p) \wedge \Box(\Box p \wedge p)) \wedge (\neg p \wedge \Diamond\Diamond\Diamond\Diamond\Diamond p)$	0.002	unsat	0.0018	unsat
$\neg(((\Box\neg p \wedge \Diamond\neg q) \vee (\Box\Diamond q \wedge \Box\Box\neg q)) \rightarrow \Box(\neg\Box p \vee \neg\Diamond q))$	0.002	unsat	0.00127	unsat

4.9.4. *Comparison Summary.* From the comparison tables 6, 7, 8 we can see that satisfiable and unsatisfiable formulas have been assigned correct results consistently in Spartacus software and our implementations. We used random non trivial formulas for all three axioms; this helped us in ensuring that our implementations work correctly and that a sufficient number of combinations of alpha, beta, delta and gamma formulas have been considered.

We also compared the time it took for different formulas to be checked for satisfiability. Our implementations were not designed for speed and efficiency as we did not focus on this but on accuracy. However, the results show that solvers for logic K, T and K4 in fact work faster than Spartacus for a number of different formulas. This is a great result since the Spartacus implementation was in fact designed for efficiency and speed.

We note in addition that our solvers are taking time in generating graphical models, which Spartacus does not do. Without our graphical output our solvers would compete even more favourably with Spartacus. Another significant difference is that Spartacus only searches for a single model in which the formula is true, whereas our solvers return all possible models. However, if we were to modify our implementations so that they only return one model then the time taken to verify formulas would again be a lot shorter. Lastly, we can observe that our K4 solver, which implements a loop checker (algorithm 6b) works faster than Spartacus for the various formulas. This in spite of the fact that the loop checker is computationally expensive because every time a new node is to be added, the loop checker scans through all the nodes and compares the formulas.

4.10. **Critical Evaluation.**

For the purpose of this project report, we are required to critically evaluate our software and the results that we achieved. We start this procedure by referring back to the aims and objectives of this project. We will then focus on results and testing procedures that were performed including comparisons against other software. Lastly, we will point out the things that could be improved in future developments.

4.10.1. *Evaluation- Aims and Objectives.* The key aims and objectives that were initially set out for this project have been fulfilled. We have a fully functional theorem prover for a modal logic K that makes use of tableaux methodologies. It implements a modal formula parser and also a consistency checker. Modal logic K is the simplest one, therefore termination and output of a finite model was not a problem. We expanded the K solver into stronger logics T, KB, B, K4, S4 and S5. Although the S5 solver was a significant extension, we still used the K solver as the basic building block. Solvers for logics T, KB and B follow directly the implementation of the K solver. For each of them we had to ensure that the additional restriction such as reflexivity or symmetricity on the Kripke frame was dealt with appropriately by a specific function. The satisfiability decision problem for K, T, KB, B and S5 is NP-Complete, so these problems did not require additional procedures in order to terminate and return a finite model. We now reach solvers for logic K4 and S4. These axioms have a higher complexity which is in

fact PSPACE-Complete. The problems we encountered during the development phase were that both solvers did not terminate for most of the formulas. This was because of the transitive relation which caused models to be expanded to infinity. To fix this, a loop checker was added to the K4 and S4 solvers. Such a modification allowed for termination of the program. The down side of the loop checker is that it is computationally expensive to run; for very large graphs it will take a lot of computations to verify whether the same node already exists in the graph or not. In terms of functional software, we delivered seven fully working solvers. They were thoroughly tested for correctness; unit tests were used to verify that small parts of the software work as intended. We also verified our results with another tableaux solver, Spartacus, which is strong evidence that our solvers are working correctly.

A secondary aim of the project was to research and to conduct a survey on the complexity of different modal logics. There are different decision problems, therefore we mainly focused on satisfiability and validity problems. The results and corresponding references were presented in table 1. As part of the research we proved that the validity problem in propositional modal logic K is decidable 2.10. For the proof we used the filtration technique which is defined in Section II. Furthermore, we have that any combinations of axioms T, S4 and S5 on top of K are decidable (see 2.11). In section 2.7 we present evidence that all axioms that we considered in this project are in fact sound and complete. Moreover, the tableaux method used to check the satisfiability or validity of the formula is also sound and complete.

The final set of general objectives was to produce a project plan, interim report, and a user manual. Each of these deliverables was completed and forms part of the final project report. Additionally, the final report includes all the research details, documentation of implemented algorithms, presentation of results and testing, and finally fully documented source code.

4.10.2. *Evaluation- Results and Testing.* The results we presented for each solver were verified to be true by a manual check and by comparison with Spartacus software (axioms K, T, and K4). We also described our implementations in detail. So, now we are going to focus on aspects that may be problematic.

Firstly, formulas that we used in this report were non trivial and had different lengths. However, if we were to consider a formula of size 1000, our solver would return models but such models could be overloaded with nodes, edges and labels. This is one of the downsides that we face when dealing with modal logic tableaux solvers. Such a disadvantage is in fact present in all implementations that are producing visual models for cases in which the formula is satisfiable. We should additionally remember that even when our models are overloaded and unreadable, we still get an output in the terminal which tells us whether a formula is satisfiable and how many models there are.

In the testing phase we manually tested formulas and we later checked whether our results are consistent with Spartacus. For this purpose we created random formulas of different sizes containing alpha, beta, delta and gamma formulas. We did not perform tests of our programs with benchmark formulas, such as these noted at http://home.inf.unibe.ch/lwb/benchmarks/benchmarks.html[BHS00]. This means that we do not know how our solvers perform with these challenging formulas. These formulas are very large in size containing at least 5000 clauses and over 10000 literals; they also differ in format to those which our solvers and Spartacus accept. Converting such very large formulas to a format accepted by our solvers would be very difficult and therefore we decided not to use them. However, it is something that can be considered in future developments.

4.10.3. *Evaluation- Future Improvements.* We already pointed out some of the disadvantages of the current design, so let's focus here on the fact that all the formulas currently need to be verified manually or against another piece of software. Both operations require a significant amount of time in order to verify whether a result is correct. So, future development should tackle this problem by implementing an internal model checking algorithm.

Lastly, in the future we could optimise our solvers so that they can evaluate the formula more quickly and efficiently. As we mentioned already one of the options would be to modify our solvers so that they output a satisfiable result once a single model is found. This could significantly improve the processing time because we will not be returning all possible models.

5. Conclusions and Future Work

5.1. **Concluding Remarks.**

In this project, we have proposed and delivered a fully functional set of propositional modal logic solvers. There are satisfiability solvers for axioms K, T, KB, B, K4, S4 and S5. Solvers allow users to verify whether an input formula is satisfiable or valid in a specific axiom. The results that are produced for each formula are in the form of textual output and also visual graphs. The process is fully automated, therefore the only action required by the user is to input the formula in a valid format.

Our implementations have been extensively tested throughout the development phase. We have used our custom set of unit tests which verified that different functions work correctly (section 6.2). The next phase of testing involved another piece of software called Spartacus. Spartacus is a tableaux prover for hybrid logic, therefore we were able to compare and verify our results against Spartacus for solvers K, T and K4. We have found consistent answers for a number of different formulas, which is a strong indication that our solvers evaluate formulas correctly. Furthermore, we measured the time it takes for our tableaux solvers to evaluate a formula and then compared it with Spartacus's execution time. It is important to note that we did not intend to implement fast and fully optimised solvers, whereas Spartacus implements optimisations for speed purposes. However, even so our tests have shown that Spartacus is slower than our solvers for a number of different formulas.

Implementations produced for this project include theorem provers for modal logic K4 and S4; these include transitive relations. These problems have PSPACE-Complete complexity, which makes them much harder problems than other axioms that we considered. In particular, it is possible to have an infinite computation that never terminates. For this reason we include in addition a loop checking algorithm. Therefore, we have K4 and S4 solvers which always terminate with a finite model if one exists.

The survey part of the project covers a large spectrum of the modal logic area. In section 2 we introduced a number of different definitions and proofs which allowed us to understand the principles. Furthermore, we summarised the results of our survey on the complexity of modal logics and then present the relative strength of covered Kripke frames. Lastly, to allow our readers to fully understand where modal logic is used we covered ways to modify it and what these modifications result in.

5.2. **Future Recommendations and Work.**

In section 4.10 we already outlined potential improvements to the software we developed. These are changes that improve the functionality and the speed of our tableaux solvers. However the software could be developed still further.

In general, there are many possibilities as to how this could be done, mainly because basic modal logic has limited applications on its own. For this reason it can be modified in various directions. One of the ways to modify modal logic is to add more modalities which will result in multi-modal logic. Also, we can derive different types of modal logic when we add extra constraints on the Kripke frame or when we combine different axioms. These are valid modifications to our existing implementations and applying them would lead us to obtaining multi-modal logics like Propositional Temporal Logic (PTL) or Computation Tree Logic (CTL) [BBR14]. The future implementation of PTL could then be used for the formal verification of computer programs.

Lastly, future work could be dedicated to creating a website which would use our current tableaux solvers. It would be an important upgrade which would allow for an easier interaction with our solvers and would be available to access via internet from any computer. This is not the case right now since we require a python interpreter and specific libraries. It could be a challenging task to develop such a website because a graphical interface will play an important part; however it is definitely a valid improvement and a future task to complete.

5.3. **Final Thoughts.**

To summarise, we will share a few final thoughts on the project in general. Throughout the duration of the project, we came across different problems related to our implementations. Nevertheless, it was a great experience to learn and develop our analytical skills along with research skills.

It was an open research project with no clear limits to indicate when the project was finished. We managed to work efficiently with a good pace which allowed our project to go as far as creating seven theorem provers.

We believe that the project fulfilled its goals and aims, providing a useful modal logic platform for solving decision problems. Furthermore, it can be expanded into more complex logics which make use of propositional modal logic. The tableaux methodology and loop checking proved to be working very well and for different formulas our algorithms proved to be even faster than Spartacus.

Finally, the project was highly successful. It was a challenging and demanding piece of research which in our opinion was well achieved. The major constraint we had was the time frame. We feel certain that we could

achieve a lot more if we had a few more months available. Preliminary what has already been done is internal model checking, but was not sufficiently developed to be presented in the report. Overall we feel that the flexibility and generals of the tableaux methods used here is of sufficient promise to motivate the further work suggested.

References

[BBR14] T.A. Beyene, M. Brockschmidt, A. Rybalchenko, CTL+FO Verification as Constraint Solving, San Jose, CA, USA, ACM, 2014.

[Bet55a] E.W. Beth, Remarks on natural deduction, Indagationes Mathematicae, 17: p. 322–325. (dedicated to Robert Feys). 1955.

[Bet55b] E.W. Beth, 'Semantic entailment and formal derivability, Med. Konkl.Nederl. Akad. v. Wetensch., Nieuwe Reeks 18(13): p. 309–342, 1955.

[Bet56] E.W. Beth, l'Existence en math'ematiques, number 10 in Collection de logique math'ematiques, s'erie A', Gauthier-Villars/Nauwelaerts, Paris/Louvain, p. 66, Beth's lectures at Sorbonne University (Paris), March 29 – April 2, 1954.

[BB06] P. Blackburn, J. van Benthem, Handbook of Modal Logic, Handbook editors: F. Wolter, Elsevier BV, 2006.

[BF03] J.C. Beall, B.C.van Fraassen, Possibilities and Paradox: An Introduction to Modal and Many-Valued Logic, 1st Edition, Oxford University Press, 2003.

[BHS00] P. Balsiger, A. Heuerding, S. Schwendimann, A Benchmark Method for the Propositional Modal Logics K, KT, S4, Kluwer Academic Publishers, Journal of Automated Reasoning 24: p. 297–317, 2000.

[BS98] W. Bibel, P.H. Schmitt (Eds.), Automated Deduction - A Basis for Applications, Volume I Foundations, Volume II Systems and Implementation Techniques, Springer-Science+ Business Media, B.V., Originally published by Kluwer Academic Publishers in 1998.

[Car87] W.A. Carnielli, Systematization of Finite Many-Valued Logics Through the Method of Tableaux, The Journal of Symbolic Logic 52 (2): p. 473–493, 1987.

[CCGH97] M.A. Castilho, L.F.del Cerro, O. Gasquet, A. Herzig, Modal tableaux with propagation rules and structural rules, Fundamenta Informaticae 32: no. 3-4, p. 281-297, 1997.

[CF08] N.B. Cocchiarella, M.A. Freund, Modal Logic: An Introduction to its Syntax and Semantics, Oxford University Press, 2008.

[Che80] B.F. Chellas, Modal Logic: An Introduction, Cambridge University Press, 1980.

[CH96] M. J. Cresswell, G.E. Hughes A New Introduction to Modal Logic, Taylor & Francis Ltd, London, United Kingdom, 1996.

[DHK07] H.v. Ditmarsch, W.v.d. Hoek, B. Kooi, Dynamic Epistemic Logic, vol. 337 of Synthese Library, Springer, 2007.

[Fit87] M.C. Fitting, Computability Theory, Semantics, and Logic Programming, Oxford Logic Guides: 13, Oxford University Press, 1987.

[Fit69] M.C. Fitting, Intuitionistic Logic Model Theory and Forcing, North Holland Publishing Company, 1969.

[GHS06] O. Gasquet, A. Herzig, M. Sahade, Terminating modal tableaux with simple completness proof, Advances in Modal Logic, vol.6, 2006.

[GKS10] D. Gotzmann, M. Kaminski, G. Smolka, Spartacus: A Tableau Prover for Hybrid Logic, In Proc. 6th Workshop on Methods for Modalities (M4M-6), ENTCS 262, p. 127-139, 2010.

[Got09] D. Gotzmann, Spartacus: A Tableau Prover for Hybrid Logic, M.Sc. Thesis, Saarland University, 2009.

[Gup11] A. Gupta, Tableaux and Proof Search in various Logics, Masters Project Report, Department of Mathematics and Statistics Indian Institute of Technology Kanpur, p. 19-23, April 2011.

[Hal95] J.Y. Halpern, The effect of bounding the number of primitive propositions and the depth of nesting on the complexity of modal logic, Artificial Intelligence 75(2):p. 361–372, 1995.

[HC96] G.E. Hughes, M. J. Cresswell, A New Introduction to Modal Logic, Routledge, 1996.

[HDFD16] J. Hunter, D. Dale, E. Firing, M. Droettboom and the matplotlib development team, 2002-2016, Available at: http://matplotlib.org/

[Hol05] B. Holen, A Reflective Theorem Prover for the Connection Calculus, Universitetet i Oslo, Institutt for informatikk, Master Thesis, 2005.

[HR99] M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, 1999.

[Hur07] P. Hurley, A Concise Introduction to Logic 10th edition, Wadsworth Publishing, p. 392, 2007.

[KPH] Y. Kazakov, I. Pratt-Hartmann, A Note on the Complexity of the Satisfiability Problem for Graded Modal Logics, Oxford, Manchester University. Available at: https://www.cs.ox.ac.uk/files/2271/gml-lics.pdf

[Kri59a] S. Kripke, A completeness theorem in modal logic, The Journal of Symbolic Logic, vol. 24, p. 1–14, 1959.

[Kri59b] S. Kripke, Semantical analysis of modal logic, The Journal of Symbolic Logic, vol.24, p. 323–324, 1959.

[Kri63] S. Kripke, Semantical analysis of modal logic I. Normal modal propositional calculi, Zeitschrift f. math. Logik und Grund. d. Math., vol. 9, p. 67–96, 1963.

[Kri65] S. Kripke, Semantical analysis of modal logic II. Non-normal modal propositional calculi, in J. W. Addison, L. Henkin and A. Tarski (eds) The Theory of Models, North-Holland, p. 206–220, 1965.

[KS09a] M. Kaminski, G. Smolka, Terminating Tableau Systems for Hybrid Logic with Difference and Converse, Journal of Logic, Language and Information 18(4), p. 437-464, 2009.

[KS09b] M. Kaminski, G. Smolka, Hybrid Tableaux for the Difference Modality, In Proc. 5th Workshop on Methods for Modalities (M4M-5),ENTCS 231, p. 241-257, 2009.

[Lad77] R.E. Ladner The computational complexity of provability in systems of modal propositinal logic*, SIAM Journal of Computing, Vol. 6, No. 3, p. 467–480, 1977.

[Mas] R. Mastop, Modal Logic for Artificial Intelligence, Course notes, Available at: www.phil.uu.nl/ rumberg/infolai/Modal_Logic.pdf

[MO98] J. Michaliszyn, J. Otop, Elementary Modal Logics over Transitive Structures, University Of Wroclaw, Imperial College London,IST Austria, ACM Mathematical Logic, 1998.

[NetX16] NetworkX developer team, 2014-2016, Available at: https://networkx.github.io/

[Ngu05] L.A. Nguyen, On the Complexity of Fragments of Modal Logics, Institute of Informatics, University of Warsaw, Advances in Modal Logic, Vol. 5, 2005.

[Plat10] A. Platzer, Lecture Notes on Decidability and Filtration, Lecture 21, April 8, 2010.

[Pnu77] A. Pnueli, The temporal logic of programs, Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), 1977.

[Pri01] G. Priest, An Introduction to Non-Classical Logic, Cambridge University Press, 2001.

[Rey13] M. Reynolds, A Faster Tableau for CTL*, Fourth International Symposium on Games, Automata, Logics and Formal Verification EPTCS 119, p. 50–63, 2013.

[Sav70] W.J. Savitch, Relationships between nondeterministic and deterministic tape complexities, Journal of Computer and System Sciences 4 (2): p. 177–192, 1970.

[Sta01] R. Stansifer, Completeness of Propositional Logic as a Program, Department of Computer Sciences, Florida Institute of Technology, Melbourne, Florida USA, March 2001.

[Sip97] M. Sipser, Michael, Section 8.3: PSPACE-completeness, Introduction to the Theory of Computation, PWS Publishing, p. 283–294, 1997.

[Zal95] E.N. Zalta, Basic Concepts in Modal Logic, Center for the Study of Language and Information, Stanford University, 1995.

6. Appendix

6.1. User and System Manual.

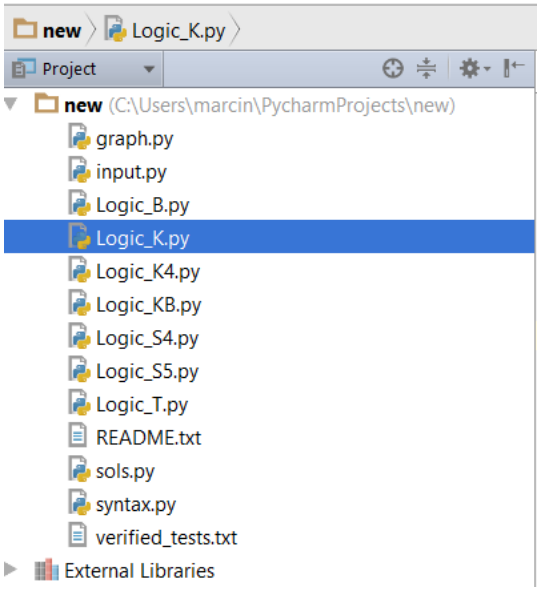
In order to run our modal solvers, Python interpretor version 2.7 is required. The software is not compatible with higher versions such as 3.X.

You can use any environment of your choice, however in this manual we suggest to use a user-friendly environment called JetBrains PyCharm Community. A free version of this software can be downloaded from the official source at: https://www.jetbrains.com/pycharm/. We are not including the configuration of the environment. Therefore, you must make sure that you download latest NetworkX and Matplotlib libraries. These are essential components which allow our solvers to solve satisfiability problem.

Next we have the following steps:

(1) After we have our environment set-up and ready to use, we need to download the source code from the public github repository https://github.com/marcincuber/modal_logic/.

(2) The next step is to import all the downloaded files into our project folder. We should have the same files available as presented in the following figure:



(3) Once we got all the files ready, we can see that there is a number of files with the name Logic_?.py where ? corresponds to the axiom in which we want to evaluate our formula. We select for example, Logic_K.py and we find the line with number 34 starting with str_psi = "". Between the "" we place our formula. The following figure shows exactly what it looks like.

```
31      '''
32          :Input String:
33      '''
34      str_psi = "~B~D(DBs ^ D(~Bt ^ Dr)) V D~(DBp > BBu) "
35
```

(4) Finally when our formula is in str_psi, we simply run the solver by clicking the green triangle, see the following figure. We will then get all the results for the specific formula.

Instructions to format the input formula.

- Set of propositional letters: $\{a, b, c, ..., x, y, z\}$
- List of connectives: $[\sim, V, \hat{}, >]$, parsed connectives get the following representations [not, or, and, imply] respectively.
- List of modalities: $[D, B]$, parsed modalities are [diamond, box] respectively.

Examples of input formulas and their representations compatible with solvers:

- Formula: $\Box(p \vee q)$, str_psi = "B(p V q)"
- Formula: $\Diamond(p \wedge q)$, str_psi = "D(p ^ q)"
- Formula: $\Diamond\Box(\neg p \rightarrow \Diamond q)$, str_psi = "$DB$(~p > Dq)"
- Formula: $\Diamond\neg\Box(\neg p \rightarrow \Diamond(q \vee \Box p))$, str_psi = "D~B(~p > D(q V Bp))"

6.2. **Unit Tests.**

Unit Tests were performed as part of the development stage. These tests were performed to verify that single functions required for solvers work correctly. In this section we will include actual code used for tests, explanations and results.

The first set of tests verified that parser converts formulas into correct format. That is a string which is an input formula to the program is converted into prefix format. So we have the following:

(1) We check whether alpha formula $(\neg\Box p \wedge \Diamond q)$ is correctly converted by parse_formula function.

```
def test_convert_form1_and():
    str_psi = "(~Bp ^ Dq) "
    psi = syntax.parse_formula(SET, str_psi)
    converted_string = ('and', ('not', ('box', 'p')), ('diamond', 'q'
        ))
    assert psi == converted_string
```

(2) We check whether beta formula (or) $(\Diamond\Box p \vee \Box\Diamond q)$ is correctly converted by parse_formula function.

```
def test_convert_form2_or():
    str_psi = "(DBp V BDq) "
    psi = syntax.parse_formula(SET, str_psi)
    converted_string = ('or', ('diamond', ('box', 'p')), ('box',('
        diamond', 'q')))
    assert psi == converted_string
```

(3) We check whether beta formula (implication) $(\Diamond\Box p \rightarrow \Box\Diamond q)$ is correctly converted by parse_formula function.

```
def test_convert_form3_imply():
    str_psi = "(DBp > BDq) "
    psi = syntax.parse_formula(SET, str_psi)
    converted_string = ('imply', ('diamond', ('box', 'p')), ('box',('
        diamond', 'q')))
    assert psi == converted_string
```

(4) We check whether formula $(\neg\Box p \wedge \Diamond q \vee s)$ is recognised as being ambiguous and therefore error is should be produced by parse_formula function.

```
def test_convert_form4_imply():
    str_psi = "(~Bp ^ Dq V s) "
    with raises(Exception) as excinfo:
        syntax.parse_formula(SET, str_psi)
    assert 'Proposition should be in place!' in str(excinfo.value)
```

The second set of unit tests verified whether a parsed function which contains alpha formulas is correctly dealt with by the recursive function.

(1) We check whether alpha formula $(\neg\Box p \wedge \Diamond q)$ is correctly expanded by recursivealpha function.

```
def test_alpha_form1():
    Sets = []
    alpha_expand = [('not', ('box', 'p')), ('diamond', 'q')]
    str_psi = "(~Bp ^ Dq) "
    psi = syntax.parse_formula(SET, str_psi)
    Sets.append(sols.recursivealpha(psi))
    assert Sets[0] == alpha_expand
```

(2) We check whether alpha formula with nested alphas $(\neg\Box(p \wedge \Diamond q) \wedge (s \wedge (q \wedge \Diamond t)))$ is correctly expanded by recursivealpha function.

```
def test_alpha_form2():
    Sets = []
    alpha_expand = [('not', ('box', ('and', 'p', ('diamond', 'q')))),
        's', 'q', ('diamond', 't')]
    str_psi = "(~B(p ^ Dq) ^ (s ^ (q ^ Dt))) "
    psi = syntax.parse_formula(SET, str_psi)
```

```
    Sets.append(sols.recursivealpha(psi))
    assert Sets[0] == alpha_expand
```

The third set of unit tests concentrates on graphs. They verify whether correct formula is placed at a correct node and whether functions expanding alpha, beta, delta and gamma formulas work correctly.

(1) We check whether formula $(\neg\Box p \wedge \Diamond q)$ is correctly expanded by recursivealpha function. Then we verify that the expanded formula is correctly place in the single node of our graph G.

```
def test_graph_form1():
    Graphs = []
    Sets = []
    G = nx.MultiDiGraph()
    str_psi = "(~Bp ^ Dq) "
    psi = syntax.parse_formula(SET, str_psi)
    Sets.append(sols.recursivealpha(psi))
    gr.create_graph_K(G, Sets)
    Graphs.append(G)
    assert len(Graphs) == 1
    assert G.node[1] == [('not', ('box', 'p')), ('diamond', 'q')]
```

(2) We check whether formula $(\Box p \wedge \Diamond q)$ is correctly expanded by recursivealpha function and then by delta_node_solve function. Delta function expands the graph by adding a new node if diamond formula is found and also deals with gamma formulas.

```
def test_graph_form2():
    Graphs = []
    Sets = []
    formulas_in = {}
    formulas_in[1] = []
    G = nx.MultiDiGraph()
    str_psi = "(Bp ^ Dq) "
    psi = syntax.parse_formula(SET, str_psi)
    Sets.append(sols.recursivealpha(psi))
    gr.create_graph_K(G, Sets)
    Graphs.append(G)
    l.delta_node_solve(G, 1 ,formulas_in)
    assert G.node[1] == [('box', 'p'), ('diamond', 'q')]
    assert G.node[2] == ['q','p']
```

(3) We check whether beta formula $(\Box p \vee \Diamond q)$ is correctly expanded beta_node_solve function. This means that graph G will be split into two graphs. One of them will contain first part of the function and the second graph will contain the second part.

```
def test_graph_form3():
    l.Graphs = []
    Sets = []
    formulas_in = {}
    formulas_in[1] = []
    G = nx.MultiDiGraph()
    str_psi = "(Bp V Dq)"
    psi = syntax.parse_formula(SET, str_psi)
    Sets.append(sols.recursivealpha(psi))
    gr.create_graph_K(G, Sets)
    l.Graphs.append(G)
    l.beta_node_solve(G, 1 ,formulas_in)
    assert len(l.Graphs) == 2
    assert l.Graphs[1].node[1] == [('diamond', 'q')]
    assert l.Graphs[0].node[1] == [('box', 'p')]
```

(4) We check whether formula $(\Box p \rightarrow (\Diamond q \wedge \Box t))$ is correctly expanded by beta_node_solve function. We then expand both graphs using delta_node_solve because in two graphs there exists delta formula. Therefore, we expect two graphs, both will contain two nodes.

```
def test_graph_form4():
    l.Graphs = []
    Sets = []
    formulas_in = {}
    formulas_in[1] = []
    G = nx.MultiDiGraph()
    str_psi = "(Bp > (Dq ^ Bt))"
    psi = syntax.parse_formula(SET, str_psi)
    Sets.append(sols.recursivealpha(psi))
    gr.create_graph_K(G, Sets)
    l.Graphs.append(G)
    l.beta_node_solve(G, 1 ,formulas_in)
    l.delta_node_solve(l.Graphs[0],1,formulas_in)
    l.delta_node_solve(l.Graphs[1],1,formulas_in)
    assert len(l.Graphs) == 2
    assert l.Graphs[0].node[1] == [('not',('box', 'p'))]
    assert l.Graphs[1].node[1] == [('diamond', 'q'),('box','t')]
    assert l.Graphs[0].node[2] == [('not', 'p')]
    assert l.Graphs[1].node[2] == ['q','t']
```

The final set of tests were related with consistency checker. In this part we have a set of formulas and function inconsistent which checks whether the set is consistent. The answer that is returned by the function is either False (consistent) or True (inconsistent).

(1) We check whether the following set of formulas is consistent $\{(p \wedge q), \neg q, p\}$. We expecting it to be inconsistent, therefore answer = True.

```
def test_consistent_form1():
    answer = True
    psi = [('and','p', 'q'), ('not', 'q'), 'p']
    result = sols.inconsistent(psi)
    assert answer == result
```

(2) We check whether the following set of formulas is consistent $\{\neg(q \vee p), p\}$. We expecting it to be inconsistent, therefore answer = True.

```
def test_consistent_form2():
    answer = True
    psi = [('not',('or','q', 'p')),  'p']
    result = sols.inconsistent(psi)
    assert answer == result
```

(3) We check whether the following set of formulas is consistent $\{\neg(p \rightarrow \neg q), \neg q\}$. We expecting it to be inconsistent, therefore answer = True.

```
def test_consistent_form3():
    answer = True
    psi = [('not',('imply', 'p', ('not','q'))), ('not','q')]
    result = sols.inconsistent(psi)
    assert answer == result
```

(4) We check whether the following set of formulas is consistent $\{\neg(\neg p \vee q), \neg(p \rightarrow q)\}$. We expecting it to be consistent, therefore answer = False.

```
def test_consistent_form4():
    answer = False
    psi = [('not',('or', ('not','p'), 'q')), ('not',('imply','p','q')
        )]
    result = sols.inconsistent(psi)
    assert answer == result
```

Please note that this is not a full list of tests that were performed. These were the initial tests that we performed on logic K solver. Lastly we show a figure which verifies that we received correct results and therefore functions work correctly.

```
platform darwin -- Python 2.7.10, pytest-2.9.0, py-1.4.31, pluggy-0.3.1 -- /System/Library/Frameworks/
Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python
cachedir: .cache
rootdir: /Users/marcincuber/Documents/PyCharm, inifile:
plugins: cov-2.2.1
collected 14 items

tests.py::test_convert_form1_and PASSED
tests.py::test_convert_form2_or PASSED
tests.py::test_convert_form3_imply PASSED
tests.py::test_convert_form4_imply PASSED
tests.py::test_alpha_form1 PASSED
tests.py::test_alpha_form2 PASSED
tests.py::test_graph_form1 PASSED
tests.py::test_graph_form2 PASSED
tests.py::test_graph_form3 PASSED
tests.py::test_graph_form4 PASSED
tests.py::test_consistent_form1 PASSED
tests.py::test_consistent_form2 PASSED
tests.py::test_consistent_form3 PASSED
tests.py::test_consistent_form4 PASSED

========================================= 14 passed in 4.93 seconds ==================================
=====
```

6.3. **Project Plan and Interim Report.**

PROJECT PLAN

by **Marcin Cuber**

15th November 2015

Project title: Modal logics research and analysis.
Supervisor's name: Prof Robin Hirsch.

Aims and objectives

Aims: Research, analyse and conduct complexity survey on the complexity (satisfiability problem) of different logics to develop deeper understanding of logic theory.
Objectives: Develop a tool (theorem prover) for Modal logic K and expand it to other logics such as T, B, S4 or S5. Implementation makes use of tableaux methodologies and outputs a Kripke frame which is a graph(model in which formula is satisfiable). Objectives are discussed on weekly basis with Robin Hirsch, therefore I am currently finishing the implementation logic K and improving my parser.

Deliverables

The project aims to produce the following deliverables:

(1) A Project Plan presenting aims and objectives of the project submit by noon Wednesday 18th November 2015.
(2) Parser for propositional modal logic and working implementation of Logic K by 20th December 2015.
(3) The Interim Report; submit document by noon Wednesday 27th January 2016, containing information about the work completed up to that point. Outline changes and work that still needs to be done.
(4) Literature review- survey concentrating on complexity of modal and temporal logic. This will be part of chapter 2 in the final report.
(5) Documentation of implemented algorithms. This will be part of chapter 3 in the final report.
(6) Deliver complete implementations of the programs. A fully documented and functional piece of software.
(7) The Final Report, complete by noon 29th April 2016. It will contain all the details about the project.

Project Plan

(1) Project start- mid-October.
(2) Literature searching and critical reviewing. This will be ongoing until the final report is finished
(3) Mid-October to December. Develop parser and algorithm for Logic K. At the moment I am finishing up my algorithm for logic K. I should be completing it earlier than expected.
(4) Report structure has been discussed with the supervisor and I will be starting my write-up very soon. Possibly at the end of November.
(5) December to mid-February. This stage will involve writing some parts of the report. However it terms of software and what is going to be implemented will be discussed in the upcoming weekly meeting.
(6) Mid-February to end of March (could take few more weeks so April will be dedicated). Work on Final Report

INTERIM REPORT

by **Marcin Cuber**

25th January 2016

Project title: Propositional Modal Logic Using Tableaux Method
Supervisor's name: Prof Robin Hirsch.

Progress to date

List of completed tasks:

(1) Fully functional parser, parses formula correctly and returns error when a formula is ambiguous.
(2) Logic K satisfiability solver- no restrictions on Kripke frame.
(3) Logic T satisfiability solver- class of all reflexive frames.
(4) Logic KB satisfiability solver- class of all symmetric frames.
(5) Logic B satisfiability solver- class of reflexive and symmetric frames.
(6) Survey on the complexity of other logics related to modal logic, this includes temporal logic and its derivations such as CTL and CTL*.

Following tasks have been started and they are partly completed:

(1) Logic K4 satisfiability solver- class of all transitive frames, most of it is finished only small changes needed to main function.
(2) Logic S4 satisfiability solver- class transitive and reflexive, partly done.
(3) Logic S5 satisfiability solver- transitive, reflexive and symmetric frames. Small part of it is done.
(4) I have made a significant progress on writing the main report. Chapter 2 of the report is almost finished. This week I will start writing chapter 3 which is dedicated to implementation details.

Remaining Work

I will now focus on finishing K4, S4 and S5 logics which could take some time to complete. If I manage to finish these I will put most of the time into report writing which I would like to finish at the end of March. In case I will have time before the project deadline I will try to implement the extension of S4 which is S4.3. This extension is additionally adding linear constraint to both transitivity and reflexivity. I am aiming to finish all my implementations up to 4 weeks time. It is possible that I will have time to implement another logic system which has a dense relation. However, it depends how quickly I manage to write my report. Regarding the report, I have started it but there are still 5 more sections to write. Nevertheless, I am ahead of my schedule and I am aiming to finish everything on time. The report is already my focus but I will be able to fully concentrate on it after I am done with the programming part of the project. My target completion date for this project is the end of March.

All the code I produced can be found here: https://github.com/marcincuber/modal_logic.

6.4. **Source Code.**

Please note that in the following code listings we use different colours for different parts of the code, they are:

- Comment: blue
- Keyword: red
- String/Char: orange
- Identifier: green

Syntax.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
ascii_setup =     [{'a','b','c','d','e','f','g','h','i','j','k','l','m','
    ↪n','o','p','q','r','s','t','u','v','w','x','y','z'},
                   # false, not, or, and, imply
                    ['F','~', 'V', '^', '>'],
                    # diamond, box
                    ('D', 'B'),
                    {('(',')'), ('[',']'), ('{','}')}
                   ]

'''
    :General class for modal language
    :we will deal with the following:
    :set of propositions {a,b,...,z}
    :list of constants
    :modal operators and its duals
    :pairs of brackets
'''
class Language:

    def __init__(self, prop = None,
                       constants = None,
                       modality = None,
                       brackets = None):
        if prop == None:
            prop = ascii_setup[0]
        self.prop = prop
        if constants == None:
            constants = ascii_setup[1]
        self.constants = constants
        if modality == None:
            modality = ascii_setup[2]
        self.modality = modality
        if brackets == None:
            brackets = ascii_setup[3]
        self.brackets = brackets


    def __getitem__(self, symbol):

        parsed_constants = ['false', 'not', 'or', 'and', 'imply']
        parsed_modalities = ['diamond', 'box']

        if symbol in self.prop:
            return symbol
        elif symbol in self.constants:
            return parsed_constants[self.constants.index(symbol)]
        elif symbol in self.modality:
            return parsed_modalities[self.modality.index(symbol)]
        else:
            return None

    def __repr__(self):
        return ('language' + '(' + str(self.prop) + ', ' +
                str(self.constants) + ', ' +
                str(self.modality) + ', ' +
                str(self.brackets) + ')')

def parse_formula(L, formula):
    return tuple_tableau(list_tableau(L, formula))

'''
    :Input: formula as a string and turns it into tableau form for
        ↪easier parsing and evaluation
'''
def list_tableau(L, formula):

    # Remove whitespace
    formula = ''.join(formula.split())
    tableau = []

    # if formula is atomic return it
    if len(formula) == 1:
        if formula in L.prop:
            return L[formula]
        else:
            raise ValueError("Proposition should be in place!")

    # Deal with formula by splitting it and generating subformulas.
    partition = [[]]
    bracket_list = []
    i = 0 #Index of current subformula.
    for part in formula:
        partition[i].append(part)
        if part in {pair[0] for pair in L.brackets}:
            bracket_list.append(part)
        elif part in {pair[1] for pair in L.brackets}:
            if not (bracket_list[-1], part) in L.brackets:
                raise ValueError("brackets are not matching up!")
            bracket_list.pop()
           # check if the bracket removed was the last one, if yes- end
                ↪ of subformula
            if bracket_list == []:
                i = i + 1
                partition.append([])
        elif bracket_list == []:
        # If we have an empty list of brackets we only deal with
            ↪propositions
            if ((part == L[0]) or # ch is bottom
               part in L.prop or
               part in L.constants[2:]): # ch is a 2-place constant
                i = i + 1
                partition.append([])
    partition = partition[:-1]

    # Delete outer brackets otherwise unary connector is the main
        ↪connector.
    if len(partition) == 1:
        if (partition[0][0], partition[0][-1]) in L.brackets:
            tableau = parse_formula(L, formula[1:-1])
        elif partition[0][0] == L.constants[1] or partition[0][0] in L.
            ↪modality:
        # partition[0][0] is in first place- neg or modality
            tableau = [L[partition[0][0]], parse_formula(L, formula[1:])
                ↪]
        else:
            raise ValueError("Not possible to partition formula")
    # If we get main connector we add it to tablueau and parse the rest
        ↪ofsubformulas
    else:
        for sub in partition:
            if (len(sub) == 1 and
                sub[0] in {const for const in (L.constants)}):
                tableau = [L[sub[0]]] + \
                          [parse_formula(L, ''.join(form)) \
                          for form in partition if not form[0] == sub
                              ↪[0]]
    return tableau

'''
    :Convert formula to tuples
'''
def tuple_tableau(format):
    if type(format) == list:
        return tuple(tuple_tableau(sub) for sub in format)
    elif type(format) == str or type(format) == tuple:
        return format
    else:
        print(format)
        raise ValueError("Error, formulas couldn't be converted to
            ↪tuples")

'''
    :Return the set of all subformulas of f.
'''
def get_subformulas(formula):
    subformulas = {formula}
    if len(formula) > 1:
        # Take 1st operand when connector is unary
        subformulas = subformulas.union(get_subformulas(formula[1]))
    if len(formula) == 3:
        # Take 2nd operand when we have binary operator
        subformulas = subformulas.union(get_subformulas(formula[2]))
    return subformulas
```

```
'''
    Close fset under subformulas.
'''
def subformula_close(formula_set):
    closed_fset = set()
    for f in formula_set:
        closed_fset = closed_fset.union(get_subformulas(f))
    return closed_fset

'''
    :Turn a input formula (list) into a string format
'''
def formula_to_string(formula):

    if len(formula) == 1:
        # The formula is atomic.
        string = formula[0]
    else:
        # handling unary operators
        if len(formula) == 2:
            if formula[0] == 'box':
                string = '[]'
            elif formula[0] == 'diamond':
                string = '<>'
            else:
                string = '~'
        else:
            string = ''

        # Deal with 1st operand
        if len(formula[1]) < 3:
            string += formula_to_string(formula[1])
        else:
            string += '(' + formula_to_string(formula[1]) + ')'

        # Deal with rest of formula when connector is binary
        if len(formula) == 3:
            # Handle operators
            if formula[0] == 'and':
                string += ' ^ '
            elif formula[0] == 'or':
                string += ' V '
            elif formula[0] == 'imply':
                string += ' -> '
            elif formula[0] == 'box':
                string += ' [] '
            elif formula[0] == 'diamond':
                string+= ' <> '

            # Deal with 2nd operand
            if len(formula[2]) < 3:
                string += formula_to_string(formula[2])
            else:
                string += '(' + formula_to_string(formula[2]) + ')'
    return string
```

Sols.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
'''
    :psi is our parsed formula in tableaux format
    :for each formula in the list deal with all alphas formulas
    :return a list which does not contain any alpas to deal with
'''
def recursivealpha(psi):
    #define constants and modalities
    parsed_constants = ['or', 'and', 'imply', 'not']
    parsed_modalities = ['diamond', 'box']

    #deal with formulas which have AND in the first position
    #scan formulas recursiverly until all formulas are dealt with
    if psi[0] == 'and':
        psi1_set = [psi[1]]
        if psi[1][0] in parsed_constants or psi[1][0] in
            ↪parsed_modalities:
            psi1_set = recursivealpha(psi[1])

        psi2_set = [psi[2]]
        if psi[2][0] in parsed_constants or psi[2][0] in
            ↪parsed_modalities:
            psi2_set = recursivealpha(psi[2])

        return psi1_set + psi2_set

    #deal with formulas which contain double negations
    elif psi[0] == 'not' and isinstance(psi[1], tuple) and psi[1][0] ==
        ↪'not':
        result = [psi[1][1]]
        if psi[1][1][0] in parsed_constants or psi[1][1][0] in
            ↪parsed_modalities:
            result = recursivealpha(psi[1][1])

        return result

    #deal with formulas with NOT(IMPLY,A,B) in which we return negated B
        ↪ and A
    elif psi[0] == 'not' and isinstance(psi[1], tuple) and psi[1][0] ==
        ↪'imply':
        psi1 = (psi[1][1])
        psi1_set = [psi1]
        if psi[1][1][0] in parsed_constants or psi[1][1][0] in
            ↪parsed_modalities:
            psi1_set = recursivealpha(psi1)

        psi2_set = [('not',psi[1][2])]
        if isinstance(psi[1][2], str):
            psi2_set = [('not',psi[1][2])]
        elif psi[1][2][0] in parsed_constants or psi[1][2][0] in
            ↪parsed_modalities:
            psi2_set = recursivealpha(('not',psi[1][2]))

        return (psi1_set + psi2_set)

    #deal with formulas with NOT(OR,A,B) in which we return negated B
        ↪and negated A
    elif psi[0] == 'not' and isinstance(psi[1], tuple) and psi[1][0] ==
        ↪'or':
        psi1 = ('not', psi[1][1])
        psi1_set = ['not',psi[1][1]]
        if psi[1][1][0] in parsed_constants or psi[1][1][0] in
            ↪parsed_modalities:
            psi1_set = recursivealpha(psi1)

        psi2 = ('not', psi[1][2])
        psi2_set = [('not', psi[1][2])]
        if psi[1][2][0] in parsed_constants or psi[1][2][0] in
            ↪parsed_modalities:
            psi2_set = recursivealpha(psi2)

        return psi1_set + psi2_set

    #if there are formulas which are not alphas, just add them to the
        ↪list
    else:
        return [psi]

'''
    :Function finds and removes inconsistent formulas from the list
'''
def inconsistent(psi):
    #initialise status to be false psi meaning that psi is consistent
    status = False;
    main_list=psi
    for i in range(0,len(psi)):
        for j in range(0,len(psi[i])):
            #we assign each formula from the list to a variable main
            main = psi[i]

            if isinstance(main, str):
                contra = ('not',main)
                if (contra in main_list) and (main in main_list):
                    status = True
                    break
            elif main[0] == 'not' and main[1][0] == 'not' and isinstance
                ↪(main[1][1],str):
                if (main[1] in main_list):
                    status = True
                    break

            elif main[0] == 'not' and main[1][0] == 'or' and isinstance(
                ↪main[1][1], tuple) and \
                main[1][1][0] == 'not' and isinstance(main[1][1][1],str)
                    ↪:
                contra = ('not',main[1][1][1])
                if (contra in main_list):
                    status = True
                    break
                elif main[0] == 'not' and main[1][0] == 'or' and
                    ↪isinstance(main[1][2], tuple) and \
```

```
    main[1][2][0] == 'not' and isinstance(main[1][2][1],str)
        ↪:
        contra = ('not',main[1][2][1])
        if (contra in main_list):
            status = True
            break
        elif main[0] == 'not' and main[1][0] == 'or' and
            ↪isinstance(main[1][2],str):
            if (main[1][2] in main_list):
                status = True
                break

elif main[0] == 'not' and main[1][0] == 'or' and isinstance(
    ↪main[1][2], tuple) and \
    main[1][2][0] == 'not' and isinstance(main[1][2][1],str)
        ↪:
        contra = ('not',main[1][2][1])
        if (contra in main_list):
            status = True
            break
        elif main[0] == 'not' and main[1][0] == 'or' and
            ↪isinstance(main[1][1], tuple) and \
            main[1][1][0] == 'not' and isinstance(main
                ↪[1][1][1],str):
            contra = ('not',main[1][1][1])
            if (contra in main_list):
                status = True
                break
            elif main[0] == 'not' and main[1][0] == 'or' and
                ↪ isinstance(main[1][1],str):
                if (main[1][1] in main_list):
                    status = True
                    break

elif main[0] == 'not' and main[1][0] == 'imply' and
    ↪isinstance(main[1][1], tuple) and \
    main[1][1][0] == 'not' and isinstance(main[1][1][1],str)
        ↪:
    print "we are here:"
    if (main[1][1][1] in main_list):
        status = True
        break
    elif main[0] == 'not' and main[1][0] == 'imply' and
        ↪isinstance(main[1][2], tuple) and \
    main[1][2][0] == 'not' and isinstance(main[1][2][1],str)
        ↪:
        if (main[1][2][1] in main_list):
            status = True
            break
        elif main[0] == 'not' and main[1][0] == 'imply' and
            ↪isinstance(main[1][2],str):
            if (main[1][2] in main_list):
                status = True
                break

elif main[0] == 'not' and main[1][0] == 'imply' and
    ↪isinstance(main[1][2], tuple) and \
    main[1][2][0] == 'not' and isinstance(main[1][2][1],str)
        ↪:
        contra = ('not',main[1][2][1])
        if (contra in main_list):
            status = True
            break
        elif main[0] == 'not' and main[1][0] == 'imply' and
            ↪isinstance(main[1][1], tuple) and \
            main[1][1][0] == 'not' and isinstance(main
                ↪[1][1][1],str):
            if (main[1][1][1] in main_list):
                status = True
                break
            elif main[0] == 'not' and main[1][0] == 'imply'
                ↪and isinstance(main[1][1],str):
                contra = ('not', main[1][1])
                if (contra in main_list):
                    status = True
                    break

elif main[0] == 'not' and main[1][0] == 'or' and isinstance(
    ↪main[1][1], str):
    if (main[1][1] in main_list):
        status = True
        break
    elif main[0] == 'not' and main[1][0] == 'or' and
        ↪isinstance(main[1][2], str):
        if (main[1][2] in main_list):
            status = True
            break

elif main[0] == 'not' and main[1][0] == 'or' and isinstance(
    ↪main[1][2], str):
    if (main[1][2] in main_list):
        status = True
        break
    elif main[0] == 'not' and main[1][0] == 'or' and
        ↪isinstance(main[1][1], str):
        if (main[1][1] in main_list):
            status = True
            break

elif main[0] == 'not' and main[1][0] == 'imply' and
    ↪isinstance(main[1][1], str):
    contra = ('not',main[1][1])
    if (contra in main_list):
        status = True
        break
    elif main[0] == 'not' and main[1][0] == 'imply' and
        ↪isinstance(main[1][2], str):
        if (main[1][2] in main_list):
            status = True
            break

elif main[0] == 'not' and main[1][0] == 'imply' and
    ↪isinstance(main[1][2], str):
    if (main[1][2] in main_list):
        status = True
        break
    elif main[0] == 'not' and main[1][0] == 'imply' and
        ↪isinstance(main[1][1], str):
        contra = ('not',main[1][1])
        if (contra in main_list):
            status = True
            break

elif main[0] == 'and' and isinstance(main[1], tuple) and
    ↪main[1][0] == 'not' and \
    isinstance(main[1][1], str):
    if (main[1][1] in main_list):
        status = True
        break
    elif main[0] == 'and' and isinstance(main[2], tuple) and
        ↪ main[2][0] == 'not' and \
    isinstance(main[2][1], str):
        if (main[2][1] in main_list):
            status = True
            break
        elif main[0] == 'and' and isinstance(main[2], str):
            contra = ('not',main[2])
            if (contra in main_list):
                status = True
                break

elif main[0] == 'and' and isinstance(main[2], tuple) and
    ↪main[2][0] == 'not' and \
    isinstance(main[2][1], str):
        if (main[2][1] in main_list):
            status = True
            break
        elif main[0] == 'and' and isinstance(main[1], tuple)
            ↪ and main[1][0] == 'not' and \
        isinstance(main[1][1], str):
            if (main[1][1] in main_list):
                status = True
                break
            elif main[0] == 'and' and isinstance(main[1],
                ↪str):
                contra = ('not',main[1])
                if (contra in main_list):
                    status = True
                    break


elif main[0] == 'and' and isinstance(main[1], str):
    contra = ('not',main[1])
    if (contra in main_list):
        status = True
        break
    elif main[0] == 'and' and isinstance(main[2], str):
        contra = ('not',main[2])
        if (contra in main_list):
            status = True
            break
```

```
            elif main[0] == 'and' and isinstance(main[2], str):
                contra = ('not',main[2])
                if (contra in main_list):
                    status = True
                    break
                elif main[0] == 'and' and isinstance(main[1], str):
                    contra = ('not',main[1])
                    if (contra in main_list):
                        status = True
                        break
    return status
```

Graph.py file:

```
__author__ = 'marcincuber'
import networkx as nx
import matplotlib.pyplot as plt
import syntax
'''
    :Logic K, T, KB, B, K4, S4, S5- graph initialisation
'''
def create_graph_K(G,Sets):
    #dictionary with labels = value assigned to each world
    custom_labels={}
    #dictionary with sizes for each node
    custom_node_sizes={}
    node_colours=['y']

    for i in range(0, len(Sets)):
        #To each label assign correspoding set with formulas
        custom_labels[i+1] = Sets[i]
        G.node[i+1] = Sets[i]
        custom_node_sizes[i+1] = 5000
        node_colours.append('b')

    #draw the graph on circular layout with colours, labels, sizes
        ↪included.
    nx.draw(G, nx.circular_layout(G),node_color=node_colours, labels=
        ↪custom_labels, node_size=1000, with_labels = True)

    #plot the graph and display the figure
    plt.show()

def final_graphs(Graphs,psi):
    if Graphs == []:
        print "There are no models for the input formula: ", (syntax.
            ↪formula_to_string(psi))
        print "So the the negation of it : ", "~(",(syntax.
            ↪formula_to_string(psi)), ") is valid."

    else:
        for i in range(0,len(Graphs)):
            graph = Graphs[i]

            custom_labels={}
            node_colours=['y']
            for node in graph.nodes():
                custom_labels[node] = graph.node[node]
                node_colours.append('c')

            nx.draw(Graphs[i], nx.circular_layout(Graphs[i]),  node_size
                ↪=1500, with_labels=True, labels = custom_labels,
                ↪node_color=node_colours)
            #show with custom labels
            fig_name = "graph" + str(i) + ".png"

            plt.savefig(fig_name)
            plt.show()

        print "Satisfiable models have been displayed."
        if len(Graphs) == 1:
            print "You have ",len(Graphs), " valid model."
        else:
            print "You have ",len(Graphs), " valid models."
        print "Your provided formula is: ", (syntax.formula_to_string(
            ↪psi))
        print "Pictures of the graphs have been saves as: graph0.png,
            ↪graph1.png etc."
```

Logic_K.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-

"""
    :Modal Logic K- no restrictions on the frame
"""
import syntax
import sols
import graph as gr
import networkx as nx
import copy
import time
from collections import OrderedDict

#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "(Bp ^ Dq)"
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

gr.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list
'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
                            set.append(prop)
            elif isinstance(value_list[i], str):
                set.append(value_list[i])
        graph.node[node] = remove_duplicates(set)
'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]
```

```
    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            #solve recursively alpha formulas
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
            else:
                for prop in alpha:
                    if prop not in set:
                        set.append(prop)

        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set
'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:

            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]

                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

'''
    :resolving DELTAS given a NODE in graph
'''
def delta_node_solve(graph, node, formulas_in):
    delta_list = graph.node[node]

    for i in range(len(delta_list)-1,-1,-1):
        part1 = delta_list[i][0]
        if part1 == 'diamond':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = delta_list[i][1]
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.node[node] = delta_list

                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)

        elif part1 == 'not' and delta_list[i][1][0] == 'box':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = ('not', delta_list[i][1][1])
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.node[node] = delta_list
                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
```

```
                    beta_node_solve(graph, new_node,
                        ↪formulas_in)
                elif set[j][0] == 'box':
                    if set[j][1] not in graph.node[new_node]:
                        graph.node[new_node].append(set[j][1])
                        alpha_node_solve(graph, new_node)
                        beta_node_solve(graph, new_node,
                            ↪formulas_in)

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""             Main loop iterating over graphs          """
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def main():
    num_graph = 0
    for graph in Graphs:
        formulas_in = graph_formulas[num_graph]
        status = 1;
        index = 1;
        alpha_node(graph)
        while status == 1:
            for node in range(index,len(graph.nodes())+1):

                start_length = len(graph.nodes())
                alpha_node_solve(graph,node)
                beta_node_solve(graph, node,formulas_in)
                delta_node_solve(graph, node,formulas_in)
                end_length = len(graph.nodes())
                if start_length < end_length:
                    diff = end_length - start_length
                    index = index+1
                elif index < len(graph.nodes()):
                    index = index+1
                else:
                    status = 0;
        num_graph += 1
    '''
        :finding inconsistencies in the model
    '''
    index_inconsistent =[]
    for i in range(0,len(Graphs)):
        graph = Graphs[i]
        for node in graph.nodes():
            consistent_list = graph.node[node]
            status = sols.inconsistent(consistent_list)
            if status == True:
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save all generated graphs
    '''
    gr.final_graphs(Graphs,psi)


t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

Logic_T.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
"""
    :Modal Logic T- reflexive frames
"""
import syntax
import sols
import graph as gr
import networkx as nx
import matplotlib.pyplot as plt
import copy
from collections import OrderedDict
import time

#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "Dp ^ BB~p"
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

gr.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list
'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
                            set.append(prop)
            elif isinstance(value_list[i], str):
                set.append(value_list[i])
        graph.node[node] = remove_duplicates(set)
'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]

    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
                else:
                    for prop in alpha:
                        if prop not in set:
                            set.append(prop)

        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set
```

```
'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:

            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]
                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)
'''
    :resolving DELTAS given a NODE in graph
'''
def delta_node_solve(graph, node, formulas_in):
    delta_list = graph.node[node]

    for i in range(len(delta_list)-1,-1,-1):
        part1 = delta_list[i][0]
        if part1 == 'diamond':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = delta_list[i][1]
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.node[node] = delta_list

                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)

        elif part1 == 'not' and delta_list[i][1][0] == 'box':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = ('not', delta_list[i][1][1])
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'

                graph.node[node] = delta_list
                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num]
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
'''
    :solving gammas at a NODE in graph
'''
def reflexive_gamma_node(graph, node, formulas_in):

    value_list = graph.node[node]
    size = len(graph.node[node])
    status = 1;
    index = 0;
    while status == 1:
        for i in range(index,size):
            value = value_list[i]
```

```
            if value[0] == 'box':
                formula = value[1]
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if formula not in graph.node[node]:
                        graph.node[node].append(formula)

            elif value[0] == 'not' and value[1][0] == "diamond":
                formula = ('not', value[1][1])
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if formula not in graph.node[node]:
                        graph.node[node].append(formula)
        new_size = len(graph.node[node])

        if size == new_size:
            status = 0;
        else:
            diff = new_size - size
            index = len(graph.node[node])-diff
            size = new_size

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""              Main loop iterating over graphs              """
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def main():
    num_graph = 0
    for graph in Graphs:
        formulas_in = graph_formulas[num_graph]
        status = 1;
        index = 1;
        alpha_node(graph)
        while status == 1:
            for node in range(index,len(graph.nodes())+1):

                start_length = len(graph.nodes())

                alpha_node_solve(graph,node)

                beta_node_solve(graph, node, formulas_in)

                reflexive_gamma_node(graph, node, formulas_in)

                delta_node_solve(graph, node,formulas_in)

                alpha_node_solve(graph,node)

                beta_node_solve(graph, node, formulas_in)

                reflexive_gamma_node(graph, node, formulas_in)

                delta_node_solve(graph, node,formulas_in)

                end_length = len(graph.nodes())
                if start_length < end_length:
                    diff = end_length - start_length
                    index = index+1
                elif index < len(graph.nodes()):
                    index = index+1
                else:
                    status = 0;
        num_graph += 1

    '''
        :finding inconsistencies in the model
    '''
    index_inconsistent =[]
    for i in range(0,len(Graphs)):
        graph = Graphs[i]
        for node in graph.nodes():
            consistent_list = graph.node[node]
            status = sols.inconsistent(consistent_list)
            if status == True:
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save as pictures all the exiting graphs in the list
    '''
    gr.final_graphs(Graphs,psi)

t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

Logic_KB.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
"""
    :Modal Logic KB- symmetric frames
"""
import syntax
import sols
import graph
import networkx as nx
import matplotlib.pyplot as plt
import copy
import time
from collections import OrderedDict

#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "(~B~Dp ^ DBDq) ^ B~s "
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

graph.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list
'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
```

```
                            set.append(prop)
                elif isinstance(value_list[i], str):
                    set.append(value_list[i])
            graph.node[node] = remove_duplicates(set)
'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
                else:
                    for prop in alpha:
                        if prop not in set:
                            set.append(prop)
        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set
'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:

            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]
                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                if part1[0] == 'not':
                    left_part = part1[1]
                else:
                    left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

'''
    :resolving DELTAS given a NODE in graph
'''
def delta_node_solve(graph, node, formulas_in):
    delta_list = graph.node[node]

    for i in range(len(delta_list)-1,-1,-1):
        part1 = delta_list[i][0]
        if part1 == 'diamond':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = delta_list[i][1]
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.add_edge((new_node), node) #add symmetric edge
                graph.node[node] = delta_list

                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)

        elif part1 == 'not' and delta_list[i][1][0] == 'box':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = ('not', delta_list[i][1][1])
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.add_edge((new_node), node) #add symmetric edge

                graph.node[node] = delta_list
                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
```

```
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)


'''
    :solving gammas at a NODE in graph
'''
def symmetric_gamma_node(graph, node, formulas_in):
    value_list = graph.node[node]
    size = len(value_list)
    index = 0
    for i in range(index,size):
        value = value_list[i]
        if value[0] == 'box':
            formula = value[1]
            if value not in formulas_in[node]:
                formulas_in[node].append(value)
                try:
                    graph.successors(node)
                except:
                    break
                next_node = graph.successors(node)
                for single_node in next_node:
                    if single_node < node:
                        #take the initial size of list to check whether
                            ↪it expanded
                        initial_size = len(graph.node[single_node])
                        if formula not in graph.node[single_node]:
                            graph.node[single_node].append(formula)

                        alpha_node_solve(graph,single_node)
                        beta_node_solve(graph,single_node, formulas_in)

                        final_size = len(graph.node[single_node])
                        #take diff to scan for these new entries
                        diff_size = final_size-initial_size
                        if diff_size > 0:
                            value_list_single_node_initial= graph.node[
                                ↪single_node]
                            value_list_single_node =
                                ↪value_list_single_node_initial[-
                                ↪diff_size:]
                            for value in value_list_single_node:
                                if isinstance(value,tuple) and value[0]
                                    ↪== 'box':
                                    part = value[1]
                                    if part not in graph.node[node]:
                                        graph.node[node].append(part)

                                elif isinstance(value,tuple) and value
                                    ↪[0] == 'not' and value[1][0] == '
                                    ↪diamond':
                                    part = ('not',value[1][1])
                                    if part not in graph.node[node]:
                                        graph.node[node].append(part)

                                elif isinstance(value, tuple) and value
                                    ↪[0] == 'diamond':
                                    part = value[1]
                                    new_node= graph.number_of_nodes()+1
                                    #adding new world
                                    graph.add_edge(single_node,(new_node
                                        ↪))
                                    #adding symmetric edge
                                    graph.add_edge((new_node),
                                        ↪single_node)

                                    formulas_in[new_node] = []
                                    graph.node[single_node] =
                                        ↪value_list_single_node_initial
                                        ↪
                                    graph.node[new_node] = [part]

                                    #expand new delta formulae
                                    previous = graph.predecessors(
                                        ↪new_node)
                                    for num in previous:
                                        set = graph.node[num];
                                        for j in range(0,len(set)):
                                            if set[j][0] == 'not' and
                                                ↪set[j][1][0] == '
                                                ↪diamond':
                                                formula = ('not',set[j
                                                    ↪][1][1])
                                                if formula not in graph.
                                                    ↪node[new_node]:
                                                    graph.node[new_node
                                                        ↪].append(
                                                        ↪formula)
                                                    alpha_node_solve(
                                                        ↪graph,
                                                        ↪new_node)
                                                    beta_node_solve(
                                                        ↪graph,
                                                        ↪new_node,
                                                        ↪formulas_in)
                                            elif set[j][0] == 'box':
                                                if set[j][1] not in
                                                    ↪graph.node[
                                                    ↪new_node]:
                                                    graph.node[new_node
                                                        ↪].append(set[
                                                        ↪j][1])
                                                    alpha_node_solve(
                                                        ↪graph,
                                                        ↪new_node)
                                                    beta_node_solve(
                                                        ↪graph,
                                                        ↪new_node,
                                                        ↪formulas_in)
                                elif isinstance(value,tuple) and value
                                    ↪[0] == 'not' and value[1][0] == '
                                    ↪box':
                                    part = ('not', value[1][1])
                                    new_node= graph.number_of_nodes()+1
                                    #adding new world
                                    graph.add_edge(single_node,(new_node
                                        ↪))
                                    #adding symmetric edge
                                    graph.add_edge((new_node),
                                        ↪single_node)

                                    formulas_in[new_node] = []
                                    graph.node[single_node] =
                                        ↪value_list_single_node_initial
                                        ↪
                                    graph.node[new_node] = [part]

                                    previous = graph.predecessors(
                                        ↪new_node)
                                    for num in previous:
                                        set = graph.node[num];
                                        for j in range(0,len(set)):
                                            if set[j][0] == 'not' and
                                                ↪set[j][1][0] == '
                                                ↪diamond':
                                                formula = ('not',set[j
                                                    ↪][1][1])
                                                if formula not in graph.
                                                    ↪node[new_node]:
                                                    graph.node[new_node
                                                        ↪].append(
                                                        ↪formula)
                                                    alpha_node_solve(
                                                        ↪graph,
                                                        ↪new_node)
                                                    beta_node_solve(
                                                        ↪graph,
                                                        ↪new_node,
                                                        ↪formulas_in)
                                            elif set[j][0] == 'box':
                                                if set[j][1] not in
                                                    ↪graph.node[
                                                    ↪new_node]:
```

```
                                        graph.node[new_node
                                            ↪].append(set[
                                            ↪j][1])
                                        alpha_node_solve(
                                            ↪graph,
                                            ↪new_node)
                                        beta_node_solve(
                                            ↪graph,
                                            ↪new_node,
                                            ↪formulas_in)

            alpha_node_solve(graph, node)
            beta_node_solve(graph, node, formulas_in)
            delta_node_solve(graph, node, formulas_in)

    elif value[0] == 'not' and value[1][0] == 'diamond':
        formula = ('not', value[1][1])
        if value not in formulas_in[node]:
            formulas_in[node].append(value)
            try:
                graph.successors(node)
            except:
                break
            next_node = graph.successors(node)

            for single_node in next_node:
                if single_node < node:
                    #take the initial size of list to check whether
                        ↪it expanded
                    initial_size = len(graph.node[single_node])
                    if formula not in graph.node[single_node]:
                        graph.node[single_node].append(formula)
                    alpha_node_solve(graph, single_node)
                    beta_node_solve(graph, single_node, formulas_in)
                    final_size = len(graph.node[single_node])

                    #take diff to scan for these new entries
                    diff_size = final_size-initial_size
                    if diff_size > 0:
                        value_list_single_node_initial= graph.node[
                            ↪single_node]
                        value_list_single_node =
                            ↪value_list_single_node_initial[-
                            ↪diff_size:]
                        for value in value_list_single_node:

                            if isinstance(value,tuple) and value[0]
                                ↪== 'box':
                                part = value[1]
                                if part not in graph.node[node]:
                                    graph.node[node].append(part)
                            elif isinstance(value,tuple) and value
                                ↪[0] == 'not' and value[1][0] == '
                                ↪diamond':
                                part = ('not',value[1][1])
                                if part not in graph.node[node]:
                                    graph.node[node].append(part)
                            elif isinstance(value, tuple) and value
                                ↪[0] == 'diamond':
                                part = value[1]
                                new_node= graph.number_of_nodes()+1
                                #adding new world
                                graph.add_edge(single_node,(new_node
                                    ↪))
                                #adding symmetric edge
                                graph.add_edge((new_node),
                                    ↪single_node)

                                formulas_in[new_node] = []
                                graph.node[single_node] =
                                    ↪value_list_single_node_initial
                                    ↪
                                graph.node[new_node] = [part]

                                previous = graph.predecessors(
                                    ↪new_node)
                                for num in previous:
                                    set = graph.node[num];
                                    for j in range(0,len(set)):
                                        if set[j][0] == 'not' and
                                            ↪set[j][1][0] == '
                                            ↪diamond':
                                            formula = ('not',set[j
                                                ↪][1][1])
                                            if formula not in graph.
                                                ↪node[new_node]:
                                                graph.node[new_node
                                                    ↪].append(
                                                    ↪formula)
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)
                                        elif set[j][0] == 'box':
                                            if set[j][0] not in
                                                ↪graph.node[
                                                ↪new_node]:
                                                graph.node[new_node
                                                    ↪].append(set[
                                                    ↪j][1])
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)

                            elif isinstance(value,tuple) and value
                                ↪[0] == 'not' and value[1][0] == '
                                ↪box':
                                part = ('not', value[1][1])
                                new_node= graph.number_of_nodes()+1
                                #adding new world
                                graph.add_edge(single_node,(new_node
                                    ↪))
                                #adding symmetric edge
                                graph.add_edge((new_node),
                                    ↪single_node)

                                formulas_in[new_node] = []
                                graph.node[single_node] =
                                    ↪value_list_single_node_initial
                                    ↪
                                graph.node[new_node] = [part]

                                previous = graph.predecessors(
                                    ↪new_node)
                                for num in previous:
                                    set = graph.node[num];
                                    for j in range(0,len(set)):
                                        if set[j][0] == 'not' and
                                            ↪set[j][1][0] == '
                                            ↪diamond':
                                            formula = ('not',set[j
                                                ↪][1][1])
                                            if formula not in graph.
                                                ↪node[new_node]:
                                                graph.node[new_node
                                                    ↪].append(
                                                    ↪formula)
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)
                                        elif set[j][0] == 'box':
                                            if set[j][0] not in
                                                ↪graph.node[
                                                ↪new_node]:
                                                graph.node[new_node
                                                    ↪].append(set[
                                                    ↪j][1])
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)
            alpha_node_solve(graph, node)
            beta_node_solve(graph, node, formulas_in)
            delta_node_solve(graph, node, formulas_in)
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""                 Main loop iterating over graphs                     """
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
```

```
def main():
    num_graph = 0
    for graph in Graphs:
        formulas_in = graph_formulas[num_graph]
        status = 1;
        index = 1;
        alpha_node(graph)
        while status == 1:
            for node in range(index,len(graph.nodes())+1):

                start_length = len(graph.nodes())

                alpha_node_solve(graph,node)

                beta_node_solve(graph, node, formulas_in)

                delta_node_solve(graph, node, formulas_in)

                symmetric_gamma_node(graph, node, formulas_in)

                delta_node_solve(graph, node, formulas_in)

                alpha_node_solve(graph,node)

                beta_node_solve(graph, node, formulas_in)

                delta_node_solve(graph, node, formulas_in)

                symmetric_gamma_node(graph, node, formulas_in)

                end_length = len(graph.nodes())
                if start_length < end_length:
                    diff = end_length - start_length
                    index = index+1
                elif index < len(graph.nodes()):
                    index = index+1
                else:
                    status = 0;
        num_graph += 1

    '''
        :finding inconsistencies in the model
    '''
    index_inconsistent =[]
    for i in range(0,len(Graphs)):
        graph = Graphs[i]
        for node in graph.nodes():
            consistent_list = graph.node[node]
            status = sols.inconsistent(consistent_list)
            if status == True:
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save as pictures all the exiting graphs in the list
    '''

    if Graphs == []:
        print "There are no models for the input formula: ", (syntax.
            ↪formula_to_string(psi))
        print "So the the negation of it : ", "~(",(syntax.
            ↪formula_to_string(psi)), ") is valid."

    else:
        for i in range(0,len(Graphs)):
            graph = Graphs[i]
            custom_labels={}
            node_colours=['y']
            for node in graph.nodes():
                custom_labels[node] = graph.node[node]
                node_colours.append('c')

            nx.draw(Graphs[i], nx.spring_layout(Graphs[i]),  node_size
                ↪=1500, with_labels=True, labels = custom_labels,
                ↪node_color=node_colours)
            #show with custom labels
            fig_name = "graph" + str(i) + ".png"

            plt.savefig(fig_name)
            plt.show()
            print "Satisfiable models have been displayed."
            if len(Graphs) == 1:
                print "You have ",len(Graphs), " valid model."
            else:
                print "You have ",len(Graphs), " valid models."
            print "Your provided formula is: ", (syntax.formula_to_string(
                ↪psi))
            print "Pictures of the graphs have been saves as: graph0.png,
                ↪graph1.png etc."

t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

Logic_B.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
"""
    :Modal Logic KB- symmetric and reflexive frames
"""
import syntax
import sols
import graph as gr
import networkx as nx
import copy
import time
from collections import OrderedDict

#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "B(BDr ^ DBt)"
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

gr.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list

'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
```

```
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
                            set.append(prop)
            elif isinstance(value_list[i], str):
                set.append(value_list[i])
        graph.node[node] = remove_duplicates(set)
'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]

    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
            else:
                for prop in alpha:
                    if prop not in set:
                        set.append(prop)

        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set
'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:

            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]
                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                if part1[0] == 'not':
                    left_part = part1[1]
                else:
                    left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

'''
    :resolving DELTAS given a NODE in graph
'''
def delta_node_solve(graph, node, formulas_in):
    delta_list = graph.node[node]

    for i in range(len(delta_list)-1,-1,-1):
        part1 = delta_list[i][0]
        if part1 == 'diamond':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = delta_list[i][1]
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.add_edge((new_node), node) #add symmetric edge
                graph.node[node] = delta_list

                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)

        elif part1 == 'not' and delta_list[i][1][0] == 'box':
```

```
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = ('not', delta_list[i][1][1])
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.add_edge((new_node), node) #add symmetric edge

                graph.node[node] = delta_list
                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
'''
    :solving gammas at a NODE in graph
'''
def reflexive_gamma_node(graph, node, formulas_in):
    value_list = graph.node[node]
    size = len(graph.node[node])
    status = 1;
    index = 0;
    while status == 1:
        for i in range(index,size):
            value = value_list[i]
            if value[0] == 'box':
                formula = value[1]
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if formula not in graph.node[node]:
                        graph.node[node].append(formula)

            elif value[0] == 'not' and value[1][0] == "diamond":
                formula = ('not', value[1][1])
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if formula not in graph.node[node]:
                        graph.node[node].append(formula)

        new_size = len(graph.node[node])
        if size == new_size:
            status = 0;
        else:
            diff = new_size - size
            index = len(graph.node[node])-diff
            size = new_size

'''
    :solving gammas at a NODE in graph
'''
def symmetric_gamma_node(graph, node, formulas_in):

    value_list = graph.node[node]
    size = len(value_list)
    index = 0
    for i in range(index,size):
        value = value_list[i]

        if value[0] == 'box':
            formula = value[1]
            try:
                parent = graph.predecessors(node)[0]
                if formula not in graph.node[parent]:
                    graph.node[parent].append(formula)

                    alpha_node_solve(graph,parent)
                    beta_node_solve(graph,parent,formulas_in)
                    reflexive_gamma_node(graph,parent,formulas_in)
                    delta_node_solve(graph,parent,formulas_in)
                    symmetric_gamma_node(graph,parent,formulas_in)
            except:
                continue

        if value not in formulas_in[node]:
            formulas_in[node].append(value)
            try:
                graph.successors(node)
            except:
                pass

            next_node = graph.successors(node)
            for single_node in next_node:
                if single_node < node:
                    #take the initial size of list to check whether
                        ↪it expanded
                    initial_size = len(graph.node[single_node])

                    if formula not in graph.node[single_node]:
                        graph.node[single_node].append(formula)
                    alpha_node_solve(graph,single_node)
                    beta_node_solve(graph,single_node, formulas_in)
                    reflexive_gamma_node(graph,node,formulas_in)

                    #size after appending formula
                    final_size = len(graph.node[single_node])
                    #take diff to scan for these new entries
                    diff_size = final_size-initial_size
                    if diff_size > 0:
                        value_list_single_node_initial= graph.node[
                            ↪single_node]
                        value_list_single_node =
                            ↪value_list_single_node_initial[-
                            ↪diff_size:]
                        for value in value_list_single_node:
                            if isinstance(value,tuple) and value[0]
                                ↪== 'box':
                                part = value[1]
                                if part not in graph.node[node]:
                                    graph.node[node].append(part)

                            elif isinstance(value,tuple) and value
                                ↪[0] == 'not' and value[1][0] == '
                                ↪diamond':
                                part = ('not',value[1][1])
                                if part not in graph.node[node]:
                                    graph.node[node].append(part)

                            elif isinstance(value, tuple) and value
                                ↪[0] == 'diamond':
                                part = value[1]
                                new_node= graph.number_of_nodes()+1
                                #adding new world
                                graph.add_edge(single_node,(new_node
                                    ↪))
                                #adding symmetric edge
                                graph.add_edge((new_node),
                                    ↪single_node)

                                formulas_in[new_node] = []
                                graph.node[single_node] =
                                    ↪value_list_single_node_initial
                                    ↪
                                graph.node[new_node] = [part]

                                #expand new delta formulae
                                previous = graph.predecessors(
                                    ↪new_node)
                                for num in previous:
                                    set = graph.node[num];
                                    for j in range(0,len(set)):
                                        if set[j][0] == 'not' and
                                            ↪set[j][1][0] == '
                                            ↪diamond':
                                            formula = ('not',set[j
                                                ↪][1][1])
                                            if formula not in graph.
                                                ↪node[new_node]:
                                                graph.node[new_node
                                                    ↪].append(
                                                    ↪formula)
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
```

```
                                        beta_node_solve(
                                            ↪graph,
                                            ↪new_node,
                                            ↪formulas_in)
                                    elif set[j][0] == 'box':
                                        if set[j][1] not in
                                            ↪graph.node[
                                            ↪new_node]:
                                            graph.node[new_node
                                                ↪].append(set[
                                                ↪j][1])
                                            alpha_node_solve(
                                                ↪graph,
                                                ↪new_node)
                                            beta_node_solve(
                                                ↪graph,
                                                ↪new_node,
                                                ↪formulas_in)
                            elif isinstance(value,tuple) and value
                                ↪[0] == 'not' and value[1][0] == '
                                ↪box':
                                part = ('not', value[1][1])
                                new_node= graph.number_of_nodes()+1
                                #adding new world
                                graph.add_edge(single_node,(new_node
                                    ↪))
                                #adding symmetric edge
                                graph.add_edge((new_node),
                                    ↪single_node)

                                formulas_in[new_node] = []
                                graph.node[single_node] =
                                    ↪value_list_single_node_initial
                                    ↪
                                graph.node[new_node] = [part]

                                previous = graph.predecessors(
                                    ↪new_node)
                                for num in previous:
                                    set = graph.node[num];
                                    for j in range(0,len(set)):
                                        if set[j][0] == 'not' and
                                            ↪set[j][1][0] == '
                                            ↪diamond':
                                            formula = ('not',set[j
                                                ↪][1][1])
                                            if formula not in graph.
                                                ↪node[new_node]:
                                                graph.node[new_node
                                                    ↪].append(
                                                    ↪formula)
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)
                                        elif set[j][0] == 'box':
                                            if set[j][1] not in
                                                ↪graph.node[
                                                ↪new_node]:
                                                graph.node[new_node
                                                    ↪].append(set[
                                                    ↪j][1])
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)

                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)
                delta_node_solve(graph, node, formulas_in)

        elif value[0] == 'not' and value[1][0] == 'diamond':
            formula = ('not', value[1][1])
            try:
                parent = graph.predecessors(node)[0]
                if formula not in graph.node[parent]:
                    graph.node[parent].append(formula)

                    alpha_node_solve(graph,parent)
                    beta_node_solve(graph,parent,formulas_in)
                    reflexive_gamma_node(graph,parent,formulas_in)
                    delta_node_solve(graph,parent,formulas_in)
                    symmetric_gamma_node(graph,parent,formulas_in)
            except:
                continue
        if value not in formulas_in[node]:
            formulas_in[node].append(value)
            try:
                graph.successors(node)
            except:
                pass
            next_node = graph.successors(node)

            for single_node in next_node:
                if single_node < node:
                    #take the initial size of list to check whether
                        ↪it expanded
                    initial_size = len(graph.node[single_node])
                    if formula not in graph.node[single_node]:
                        graph.node[single_node].append(formula)
                    alpha_node_solve(graph, single_node)
                    beta_node_solve(graph, single_node, formulas_in)
                    reflexive_gamma_node(graph,node,formulas_in)
                    final_size = len(graph.node[single_node])

                    #take diff to scan for these new entries
                    diff_size = final_size-initial_size
                    if diff_size > 0:
                        value_list_single_node_initial= graph.node[
                            ↪single_node]
                        value_list_single_node =
                            ↪value_list_single_node_initial[-
                            ↪diff_size:]
                        for value in value_list_single_node:
                            print "last value is", value
                            if isinstance(value,tuple) and value[0]
                                ↪== 'box':
                                part = value[1]
                                if part not in graph.node[node]:
                                    graph.node[node].append(part)
                            elif isinstance(value,tuple) and value
                                ↪[0] == 'not' and value[1][0] == '
                                ↪diamond':
                                part = ('not',value[1][1])
                                if part not in graph.node[node]:
                                    graph.node[node].append(part)
                            elif isinstance(value, tuple) and value
                                ↪[0] == 'diamond':
                                part = value[1]
                                new_node= graph.number_of_nodes()+1
                                #adding new world
                                graph.add_edge(single_node,(new_node
                                    ↪))
                                #adding symmetric edge
                                graph.add_edge((new_node),
                                    ↪single_node)

                                formulas_in[new_node] = []
                                graph.node[single_node] =
                                    ↪value_list_single_node_initial
                                    ↪
                                graph.node[new_node] = [part]

                                previous = graph.predecessors(
                                    ↪new_node)
                                for num in previous:
                                    set = graph.node[num];
                                    for j in range(0,len(set)):
                                        if set[j][0] == 'not' and
                                            ↪set[j][1][0] == '
                                            ↪diamond':
                                            formula = ('not',set[j
                                                ↪][1][1])
                                            if formula not in graph.
                                                ↪node[new_node]:
                                                graph.node[new_node
                                                    ↪].append(
                                                    ↪formula)
                                                alpha_node_solve(
                                                    ↪graph,
                                                    ↪new_node)
                                                beta_node_solve(
                                                    ↪graph,
                                                    ↪new_node,
                                                    ↪formulas_in)
                                        elif set[j][0] == 'box':
```

```
                                        if set[j][0] not in
                                            ↪graph.node[
                                            ↪new_node]:
                                            graph.node[new_node
                                                ↪].append(set[
                                                ↪j][1])
                                            alpha_node_solve(
                                                ↪graph,
                                                ↪new_node)
                                            beta_node_solve(
                                                ↪graph,
                                                ↪new_node,
                                                ↪formulas_in)

                                    elif isinstance(value,tuple) and value
                                        ↪[0] == 'not' and value[1][0] == '
                                        ↪box':
                                        part = ('not', value[1][1])
                                        new_node= graph.number_of_nodes()+1
                                        #adding new world
                                        graph.add_edge(single_node,(new_node
                                            ↪))
                                        #adding symmetric edge
                                        graph.add_edge((new_node),
                                            ↪single_node)

                                        formulas_in[new_node] = []
                                        graph.node[single_node] =
                                            ↪value_list_single_node_initial
                                            ↪
                                        graph.node[new_node] = [part]

                                        previous = graph.predecessors(
                                            ↪new_node)
                                        for num in previous:
                                            set = graph.node[num];
                                            for j in range(0,len(set)):
                                                if set[j][0] == 'not' and
                                                    ↪set[j][1][0] == '
                                                    ↪diamond':
                                                    formula = ('not',set[j
                                                        ↪][1][1])
                                                    if formula not in graph.
                                                        ↪node[new_node]:
                                                        graph.node[new_node
                                                            ↪].append(
                                                            ↪formula)
                                                        alpha_node_solve(
                                                            ↪graph,
                                                            ↪new_node)
                                                        beta_node_solve(
                                                            ↪graph,
                                                            ↪new_node,
                                                            ↪formulas_in)
                                                elif set[j][0] == 'box':
                                                    if set[j][0] not in
                                                        ↪graph.node[
                                                        ↪new_node]:
                                                        graph.node[new_node
                                                            ↪].append(set[
                                                            ↪j][1])
                                                        alpha_node_solve(
                                                            ↪graph,
                                                            ↪new_node)
                                                        beta_node_solve(
                                                            ↪graph,
                                                            ↪new_node,
                                                            ↪formulas_in)
                    alpha_node_solve(graph, node)
                    beta_node_solve(graph, node, formulas_in)
                    delta_node_solve(graph, node, formulas_in)

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""                 Main loop iterating over graphs                   """
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def main():
    num_graph = 0
    for graph in Graphs:
        formulas_in = graph_formulas[num_graph]
        status = 1;
        index = 1;
        alpha_node(graph)
        while status == 1:
            for node in range(index,len(graph.nodes())+1):
                start_length = len(graph.nodes())
                alpha_node_solve(graph,node)
                beta_node_solve(graph, node, formulas_in)
                reflexive_gamma_node(graph, node, formulas_in)
                alpha_node_solve(graph,node)
                beta_node_solve(graph, node, formulas_in)
                reflexive_gamma_node(graph, node, formulas_in)
                delta_node_solve(graph, node, formulas_in)
                symmetric_gamma_node(graph, node, formulas_in)
                alpha_node_solve(graph,node)
                beta_node_solve(graph, node, formulas_in)
                delta_node_solve(graph, node, formulas_in)
                reflexive_gamma_node(graph,node,formulas_in)
                symmetric_gamma_node(graph, node, formulas_in)
                end_length = len(graph.nodes())
                if start_length < end_length:
                    diff = end_length - start_length
                    index = index + 1
                elif index < len(graph.nodes()):
                    index = index + 1
                else:
                    status = 0;
        num_graph += 1

    '''
        :finding inconsistencies in the model
    '''
    index_inconsistent =[]
    for i in range(0,len(Graphs)):
        graph = Graphs[i]
        for node in graph.nodes():
            consistent_list = graph.node[node]
            status = sols.inconsistent(consistent_list)
            if status == True:
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save as pictures all the exiting graphs in the list
    '''
    gr.final_graphs(Graphs,psi)

t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

Logic_K4.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
"""
    :Modal Logic K4- all transitive frames
"""
import syntax
import sols
import graph as gr
import networkx as nx
import copy
from collections import OrderedDict
import time
#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
```

```
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "Ds ^ (BDu ^ DDq) "
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

gr.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list
'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
                            set.append(prop)
            elif isinstance(value_list[i], str):
                set.append(value_list[i])
        graph.node[node] = remove_duplicates(set)
'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]

    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
                else:
                    for prop in alpha:
                        if prop not in set:
                            set.append(prop)

        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set
'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:

            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]
                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                if part1[0] == 'not':
                    left_part = part1[1]
                else:
                    left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
```

```
                for graph in Graphs:
                    alpha_node(graph)
'''
    :resolving DELTAS given a NODE in graph
'''
def delta_node_solve(graph, node, formulas_in):
    delta_list = graph.node[node]

    for i in range(len(delta_list)-1,-1,-1):
        part1 = delta_list[i][0]
        if part1 == 'diamond':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = delta_list[i][1]
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.node[node] = delta_list

                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)
                transitive_gamma_node(graph, new_node, formulas_in)
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)

        elif part1 == 'not' and delta_list[i][1][0] == 'box':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = ('not', delta_list[i][1][1])
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'

                graph.node[node] = delta_list
                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)
                transitive_gamma_node(graph, new_node, formulas_in)
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)

def transitive_gamma_node(graph, node, formulas_in):
    parent = graph.predecessors(node)
    status = True
    if len(parent) == 0: #check if predecessor exists
        return None

    else:
        previous = graph.predecessors(parent[0])
        while status == True:
            if len(previous) == 0:
                status = False
                return None
            else:
                for num_node in previous:
                    graph.add_edge(num_node,node)
                    set = graph.node[num_node]
                    for i in range(0,len(set)):
                        if set[i][0] == 'not' and set[i][1][0] == '
                            ↪diamond':
                            formula = ('not',set[i][1][1])
                            if set[i] not in formulas_in[node]:
                                formulas_in[node].append(set[i])
                                if formula not in graph.node[node]:
                                    graph.node[node].append(formula)
                                alpha_node_solve(graph, node)
                                beta_node_solve(graph,node,formulas_in)
                        elif set[i][0] == 'box':
                            if set[i] not in formulas_in[node]:
                                formulas_in[node].append(set[i])
                                if set[i][1] not in graph.node[node]:
                                    graph.node[node].append(set[i][1])
                                alpha_node_solve(graph, node)
                                beta_node_solve(graph,node,formulas_in)
                    previous = graph.predecessors(previous[0])
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""              Main loop iterating over graphs          """
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def main():
    num_graph = 0
    for graph in Graphs:
        #dictionary of nodes: ticked formulas
        formulas_in = graph_formulas[num_graph]

        #initialise loop parameters
        status = 1;
        index = 1;
        #initial solve for alpha
        alpha_node(graph)

        #iterate over all nodes and all formulas inside them
        while status == 1:
            length = len(graph.nodes())+1
            for node in range(index,length):
                #initial number of nodes
                start_length = len(graph.nodes())

                #set internal status to False
                status_inside = False

                #verify whether node exists
                try:
                    value_node = graph.node[node]
                except:
                    #move on if it does not exists
                    status_inside = True
                    pass

                #when internal status is true add edge between them
                if status_inside == True:
                    pass

                else:
                    alpha_node_solve(graph,node)
                    beta_node_solve(graph, node, formulas_in)
                    delta_node_solve(graph,node,formulas_in)
                    alpha_node_solve(graph,node)
                    beta_node_solve(graph, node, formulas_in)
                    delta_node_solve(graph, node, formulas_in)

                    #new variable to store current node
                    current_node = node
                    #number of node at this stage
                    new_num_nodes = len(graph.nodes())

                    #deal with new nodes and compare them with existing
                        ↪ones
                    if new_num_nodes-current_node > 0:
```

```
for i in range(new_num_nodes,current_node,-1):
    #if the last new node is the same as
    try:
        if graph.node[current_node] == graph.
            ↪node[new_num_nodes] and ((
            ↪current_node,new_num_nodes) in
            ↪graph.edges()):
            graph.remove_node(new_num_nodes)
            new_num_nodes = len(graph.nodes())
            pass
        for all_nodes in range(1,new_num_nodes
            ↪-1):
            if (graph.node[new_num_nodes]==graph
                ↪.node[all_nodes]) and ((
                ↪all_nodes,new_num_nodes) not
                ↪in graph.edges()):
                graph.add_edge(current_node,
                    ↪all_nodes)
                graph.remove_node(new_num_nodes)
                new_num_nodes = len(graph.nodes
                    ↪())
                break
    except:
        pass

    try:
        temp_node = graph.node[i]
    except:
        pass

    status_inside2 = False
    for exist in range(1,current_node):
        if temp_node == graph.node[exist] and
            ↪temp_node != exist:
            #if the same node exists set status
                ↪to true and exit the loop
            status_inside2 = True
            break

    #deal with the node that already exists
    if status_inside2 == True:
        #remove the new_node which already
            ↪exists
        try:
            graph.remove_node(i)
        except:
            continue
        #add an edge from the node to existing
            ↪node
        graph.add_edge(current_node,exist)

        #get currrent list of nodes in the graph
        list_of_nodes = graph.nodes()

        #check nodes that have id > than deleted
            ↪ node and change ids, ids cannot
            ↪have gaps in numbering
        new_counter = i
        temp_value = 0
        for sig_node in list_of_nodes:
            #when id of the node is greater than
                ↪ the deleted one deal with it
            if (sig_node > new_counter+
                ↪temp_value) and ((new_counter
                ↪+temp_value) not in
                ↪list_of_nodes):
                #add missing node
                try:
                    graph.add_node(new_counter+
                        ↪temp_value)
                    #assign set of formulas to a
                        ↪ new node
                    graph.node[new_counter+
                        ↪temp_value] = graph.
                        ↪node[sig_node]
                except:
                    graph.remove_node(
                        ↪new_counter+
                        ↪temp_value)
                    break

                #scan predecessor of the higher
                    ↪node and deal with edges
                predecessor = graph.predecessors
                    ↪(sig_node)
                for pred in predecessor:
                    graph.add_edge(pred,
                        ↪new_counter+
                        ↪temp_value)

                #after we dealt with edges we
                    ↪can remove the higher
                    ↪order node
                graph.remove_node(sig_node)

                #get the current list of nodes
                list_of_nodes = graph.nodes()

                # increment temp_value in case
                    ↪there are more nodes in
                    ↪the list
                temp_value += 1

            end_length = len(graph.nodes())
            if start_length < end_length:
                diff = end_length - start_length
                index = index+1
            elif index < len(graph.nodes()):
                index = index+1
            else:
                status = 0;

        #increment number of graphs to get correct list with used
            ↪formulas
        num_graph += 1

    '''
        :finding inconsistencies in the model
    '''
    index_inconsistent =[]
    for i in range(0,len(Graphs)):
        graph = Graphs[i]
        for node in graph.nodes():
            consistent_list = graph.node[node]
            status = sols.inconsistent(consistent_list)
            if status == True:
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save as pictures all the exiting graphs in the list
    '''
    gr.final_graphs(Graphs,psi)

t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

Logic_S4.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
"""
    :Modal Logic S4- transitive and reflexive frames
"""
import syntax
import sols
import graph as gr
import networkx as nx
import copy
from collections import OrderedDict
import time
#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
```

```
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "Dp ^ BDp"
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

gr.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list
'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
                            set.append(prop)
            elif isinstance(value_list[i], str):
                set.append(value_list[i])
        graph.node[node] = remove_duplicates(set)

'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]

    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
            else:
                for prop in alpha:
                    if prop not in set:
                        set.append(prop)
        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set

'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:

            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]
                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                formulas_in[node].append(part1)
                copy_formulas_in = copy.deepcopy(formulas_in)
                copy_formulas_in[node].append(part2)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                if part1[0] == 'not':
                    left_part = part1[1]
                else:
                    left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)
```

```
'''
    :resolving DELTAS given a NODE in graph
'''
def delta_node_solve(graph, node, formulas_in):
    delta_list = graph.node[node]

    for i in range(len(delta_list)-1,-1,-1):
        part1 = delta_list[i][0]
        if part1 == 'diamond':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = delta_list[i][1]
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'
                graph.node[node] = delta_list

                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)
                transitive_gamma_node(graph, new_node, formulas_in)
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                                reflexive_gamma_node(graph,node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                                reflexive_gamma_node(graph,node,
                                    ↪formulas_in)

        elif part1 == 'not' and delta_list[i][1][0] == 'box':
            sub = delta_list[i]
            if sub not in formulas_in[node]:
                formulas_in[node].append(sub)
                part2 = ('not', delta_list[i][1][1])
                new_node= graph.number_of_nodes()+1
                graph.add_edge(node,(new_node)) #adding new world and
                    ↪relation Rxx'

                graph.node[node] = delta_list
                graph.node[new_node] = [part2]
                formulas_in[new_node] = []
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)
                transitive_gamma_node(graph, new_node, formulas_in)
                alpha_node_solve(graph, node)
                beta_node_solve(graph, node, formulas_in)

                previous = graph.predecessors(new_node)
                for num in previous:
                    set = graph.node[num];
                    for j in range(0,len(set)):
                        if set[j][0] == 'not' and set[j][1][0] == '
                            ↪diamond':
                            formula = ('not',set[j][1][1])
                            if formula not in graph.node[new_node]:
                                graph.node[new_node].append(formula)
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                                reflexive_gamma_node(graph,node,
                                    ↪formulas_in)
                        elif set[j][0] == 'box':
                            if set[j][1] not in graph.node[new_node]:
                                graph.node[new_node].append(set[j][1])
                                alpha_node_solve(graph, new_node)
                                beta_node_solve(graph, new_node,
                                    ↪formulas_in)
                                reflexive_gamma_node(graph,node,
                                    ↪formulas_in)
        alpha_node_solve(graph, node)
        beta_node_solve(graph, node, formulas_in)
'''
    :solving gammas at a NODE in graph
'''
def reflexive_gamma_node(graph, node, formulas_in):

    value_list = graph.node[node]
    size = len(graph.node[node])
    status = 1;
    index = 0;
    while status == 1:
        for i in range(index,size):
            value = value_list[i]
            if value[0] == 'box':
                formula = value[1]
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if formula not in graph.node[node]:
                        graph.node[node].append(formula)

            elif value[0] == 'not' and value[1][0] == "diamond":
                formula = ('not', value[1][1])
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if formula not in graph.node[node]:
                        graph.node[node].append(formula)

        new_size = len(graph.node[node])

        if size == new_size:
            status = 0;
        else:
            diff = new_size - size
            index = len(graph.node[node])-diff
            size = new_size

def transitive_gamma_node(graph, node, formulas_in):

    parent = graph.predecessors(node)
    status = True
    if len(parent) == 0: #check if predecessor exists
        return None

    else:
        previous = graph.predecessors(parent[0])
        while status == True:
            if len(previous) == 0:
                status = False
                return None
            else:
                for num_node in previous:
                    graph.add_edge(num_node,node)
                    set = graph.node[num_node]
                    for i in range(0,len(set)):
                        if set[i][0] == 'not' and set[i][1][0] == '
                            ↪diamond':
                            formula = ('not',set[i][1][1])
                            if set[i] not in formulas_in[node]:
                                formulas_in[node].append(set[i])
                                if formula not in graph.node[node]:
                                    graph.node[node].append(formula)
                            alpha_node_solve(graph, node)
                            beta_node_solve(graph,node,formulas_in)
                        elif set[i][0] == 'box':
                            if set[i] not in formulas_in[node]:
                                formulas_in[node].append(set[i])
                                if set[i][1] not in graph.node[node]:
                                    graph.node[node].append(set[i][1])
                            alpha_node_solve(graph, node)
                            beta_node_solve(graph,node,formulas_in)
                previous = graph.predecessors(previous[0])
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""              Main loop iterating over graphs              """
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def main():
    num_graph = 0
    for graph in Graphs:
        #dictionary of nodes: ticked formulas
        formulas_in = graph_formulas[num_graph]
```

```
#initialise loop parameters
status = 1;
index = 1;
#initial solve for alpha
alpha_node(graph)

#iterate over all nodes and all formulas inside them
while status == 1:
    length = len(graph.nodes())+1
    for node in range(index,length):
        #initial number of nodes
        start_length = len(graph.nodes())

        #set internal status to False
        status_inside = False

        #verify whether node exists
        try:
            value_node = graph.node[node]
        except:
            #move on if it does not exists
            status_inside = True
            pass

        #when internal status is true add edge between them
        if status_inside == True:
            pass

        else:
            alpha_node_solve(graph,node)
            beta_node_solve(graph, node, formulas_in)
            reflexive_gamma_node(graph, node, formulas_in)
            alpha_node_solve(graph,node)
            beta_node_solve(graph, node, formulas_in)
            delta_node_solve(graph, node, formulas_in)

            #new variable to store current node
            current_node = node
            #number of node at this stage
            new_num_nodes = len(graph.nodes())

            #deal with new nodes and compare them with existing
                ↪ones
            if new_num_nodes-current_node > 0:
                for i in range(new_num_nodes,current_node,-1):
                    #if the last new node is the same as
                    try:
                        if graph.node[current_node] == graph.
                            ↪node[new_num_nodes] and ((
                            ↪current_node,new_num_nodes) in
                            ↪graph.edges()):
                            graph.remove_node(new_num_nodes)
                            new_num_nodes = len(graph.nodes())
                            pass
                        for all_nodes in range(1,new_num_nodes
                            ↪-1):
                            if (graph.node[new_num_nodes]==graph
                                ↪.node[all_nodes]) and ((
                                ↪all_nodes,new_num_nodes) not
                                ↪in graph.edges()):
                                graph.add_edge(current_node,
                                    ↪all_nodes)
                                graph.remove_node(new_num_nodes)
                                new_num_nodes = len(graph.nodes
                                    ↪())
                                break
                    except:
                        pass

                    try:
                        temp_node = graph.node[i]
                    except:
                        pass

                    status_inside2 = False
                    for exist in range(1,current_node):
                        if temp_node == graph.node[exist] and
                            ↪temp_node != exist:
                            #if the same node exists set status
                                ↪to true and exit the loop
                            status_inside2 = True
                            break

                    #deal with the node that already exists
                    if status_inside2 == True:
                        #remove the new_node which already
                            ↪exists
                        try:
                            graph.remove_node(i)
                        except:
                            continue
                        #add an edge from the node to existing
                            ↪node
                        graph.add_edge(current_node,exist)
                        #graph.add_edge(i,exist)

                        #get currrent list of nodes in the graph
                        list_of_nodes = graph.nodes()

                        #check nodes that have id > than deleted
                            ↪ node and change ids, ids cannot
                            ↪have gaps in numbering
                        new_counter = i
                        temp_value = 0
                        for sig_node in list_of_nodes:
                            #when id of the node is greater than
                                ↪ the deleted one deal with it
                            if (sig_node > new_counter+
                                ↪temp_value) and ((new_counter
                                ↪+temp_value) not in
                                ↪list_of_nodes):
                                #add missing node
                                try:
                                    graph.add_node(new_counter+
                                        ↪temp_value)
                                    #assign set of formulas to a
                                        ↪ new node
                                    graph.node[new_counter+
                                        ↪temp_value] = graph.
                                        ↪node[sig_node]
                                except:
                                    graph.remove_node(
                                        ↪new_counter+
                                        ↪temp_value)
                                    break

                                #scan predecessor of the higher
                                    ↪node and deal with edges
                                predecessor = graph.predecessors
                                    ↪(sig_node)

                                for pred in predecessor:
                                    graph.add_edge(pred,
                                        ↪new_counter+
                                        ↪temp_value)

                                #after we dealt with edges we
                                    ↪can remove the higher
                                    ↪order node
                                graph.remove_node(sig_node)

                                #get the current list of nodes
                                list_of_nodes = graph.nodes()

                                # increment temp_value in case
                                    ↪there are more nodes in
                                    ↪the list
                                temp_value += 1
                    end_length = len(graph.nodes())
                    if start_length < end_length:
                        diff = end_length - start_length
                        index = index+1
                    elif index < len(graph.nodes()):
                        index = index+1
                    else:
                        status = 0;

        #increment number of graphs to get correct list with used
            ↪formulas
        num_graph += 1

'''
    :finding inconsistencies in the model
'''
index_inconsistent =[]
for i in range(0,len(Graphs)):
    graph = Graphs[i]
    for node in graph.nodes():
        consistent_list = graph.node[node]
        status = sols.inconsistent(consistent_list)
        if status == True:
```

```
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save as pictures all the exiting graphs in the list
    '''
    gr.final_graphs(Graphs,psi)

t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

Logic_S5.py file:

```
__author__ = 'marcincuber'
# -*- coding: utf-8 -*-
"""
    :Modal Logic S5- symmetric, reflexive and transitive frames-
        ↪equivalence relation
"""
import syntax
import sols
import graph as gr
import networkx as nx
import copy
from collections import OrderedDict
import time

#import symbols
SET = syntax.Language(*syntax.ascii_setup)

'''
    :Arrays to store the number of worlds and sets that correspond to
        ↪each world
'''
Graphs = [] #initilise empty list of graphs
Sets = [] #initilise list to store formulas which will be available in
    ↪each world

graph_formulas = [] #list of dictionaries-used formulas in node for
    ↪graph
formulas = {} #single dictionary
formulas[1] = [] #first list for node 1
graph_formulas.append(formulas)#add it to list of dictionaries

'''
    :Input String:
'''
str_psi = "((~DDB~p V (~BDp ^ DDq)) > (D~BDs ^ D~Bt)) "
print "formula input: ", (str_psi)

'''
    :Parsed string into tuple and list
'''
psi = syntax.parse_formula(SET, str_psi)
Sets.append(sols.recursivealpha(psi))

'''
    :creating initial graph
'''
G = nx.MultiDiGraph()
uniq_Sets = [list(OrderedDict.fromkeys(l)) for l in Sets]

gr.create_graph_K(G,uniq_Sets)
Graphs.append(G)

'''
    :functions to remove duplicates from the list
'''
def remove_duplicates(lista):
    return list(set(lista))

def remove_dups_graph(graph):
    for node in graph.nodes():
        value_list = graph.node[node]
        unique_list = remove_duplicates(value_list)
        graph.node[node] = unique_list
'''
    :resolving ALPHAS given a GRAPH
'''
def alpha_node(graph):
    for node in graph.nodes():
        set = []
        value_list = graph.node[node]
        for i in range(0,len(value_list)):
            if isinstance(value_list[i], tuple):
                alpha = sols.recursivealpha(value_list[i])
                for j in alpha:
                    if isinstance(j, tuple):
                        if j not in set:
                            set.append(j)
                    else:
                        for prop in alpha:
                            set.append(prop)
            elif isinstance(value_list[i], str):
                set.append(value_list[i])
        graph.node[node] = remove_duplicates(set)
'''
    :resolving ALPHAS given a NODE in graph
'''
def alpha_node_solve(graph,node):
    set = [] # array to store expanded alphas
    value_list = graph.node[node]

    for i in range(0,len(value_list)):
        if isinstance(value_list[i], tuple):
            alpha = sols.recursivealpha(value_list[i])
            for j in alpha:
                if isinstance(j, tuple):
                    if j not in set:
                        set.append(j)
            else:
                for prop in alpha:
                    if prop not in set:
                        set.append(prop)

        elif isinstance(value_list[i], str):
            if value_list[i] not in set:
                set.append(value_list[i])
    graph.node[node] = set
'''
    :resolving BETAS given a NODE in graph
'''
def beta_node_solve(graph, node, formulas_in):
    value_list = graph.node[node]
    for i in range(0,len(value_list)):
        value = value_list[i]
        if value not in formulas_in[node]:
            if value[0] =='or':
                part1 = value[1]
                part2 = value[2]
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'or':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(part1)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'not' and value[1][0] == 'and':
                part1 = value[1][1]
                part2 = value[1][2]
                left_part = ('not',part1)
                right_part = ('not',part2)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
```

```
                graph.node[node].append(left_part)
                comp2.node[node].append(right_part)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value[0] == 'imply':
                part1 = value[1]
                part2 = value[2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node].remove(value)
                comp2.node[node].remove(value)
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in[node].append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)

            elif value_list[i] == 'imply':
                part1 = value_list[i+1]
                part2 = value_list[i+2]
                left_part = ('not',part1)
                comp2 = graph.copy()
                graph.node[node] = []
                comp2.node[node] = []
                graph.node[node].append(left_part)
                comp2.node[node].append(part2)
                Graphs.append(comp2)
                formulas_in.append(value)
                copy_formulas_in = copy.deepcopy(formulas_in)
                graph_formulas.append(copy_formulas_in)
                for graph in Graphs:
                    alpha_node(graph)


    '''
        :resolving DELTAS given a NODE in graph
    '''
    def delta_node_solve(graph, node, formulas_in):
        delta_list = graph.node[node]

        for i in range(len(delta_list)-1,-1,-1):
            part1 = delta_list[i][0]
            if part1 == 'diamond':
                sub = delta_list[i]
                part2 = delta_list[i][1]

                if (sub not in formulas_in[node]) and (part2 not in graph.
                    ↪node[node]):
                    formulas_in[node].append(sub)
                    status = False
                    for node_int in graph.nodes():
                        if (sub in formulas_in[node_int]) and (node !=
                            ↪node_int):
                            status = True
                            break
                        if part2 in graph.node[node_int]:
                            status = True
                            break
                    if status == False:
                        new_node= graph.number_of_nodes()+1
                        graph.node[new_node] = [part2]
                        formulas_in[new_node] = []
                        alpha_node_solve(graph, node)
                        beta_node_solve(graph, node, formulas_in)

                        for single_node in graph.nodes():
                            if single_node != new_node:
                                set = graph.node[single_node]
                                for j in range(0,len(set)):
                                    if set[j][0] == 'not' and set[j][1][0]
                                        ↪== 'diamond' and set[j][1][1][0]
                                        ↪== 'box':
                                        pass
                                    elif set[j][0] == 'box' and set[j][1][0]
                                        ↪ == 'diamond':
                                        pass
                                    elif set[j][0] == 'not' and set[j][1][0]
                                        ↪ == 'diamond':
                                        formula = ('not',set[j][1][1])
                                        if formula not in graph.node[
                                            ↪new_node]:
                                            graph.node[new_node].append(
                                                ↪formula)
                                            alpha_node_solve(graph, new_node
                                                ↪)
                                            beta_node_solve(graph, new_node,
                                                ↪ formulas_in)
                                    elif set[j][0] == 'box':
                                        if set[j][1] not in graph.node[
                                            ↪new_node]:
                                            graph.node[new_node].append(set[
                                                ↪j][1])
                                            alpha_node_solve(graph, new_node
                                                ↪)
                                            beta_node_solve(graph, new_node,
                                                ↪ formulas_in)

            elif part1 == 'not' and delta_list[i][1][0] == 'box':
                sub = delta_list[i]
                part2 = ('not', delta_list[i][1][1])
                if (sub not in formulas_in[node]) and (part2 not in graph.
                    ↪node[node]):
                    formulas_in[node].append(sub)
                    status = False
                    for node_int in graph.nodes():
                        if (sub in formulas_in[node_int]) and (node !=
                            ↪node_int):
                            status = True
                            break
                        if part2 in graph.node[node_int]:
                            status = True
                            break
                    if status == False:
                        new_node= graph.number_of_nodes()+1
                        graph.node[new_node] = [part2]
                        formulas_in[new_node] = []
                        alpha_node_solve(graph, node)
                        beta_node_solve(graph, node, formulas_in)

                        for single_node in graph.nodes():
                            if single_node != new_node:
                                set = graph.node[single_node]
                                for j in range(0,len(set)):
                                    if set[j][0] == 'not' and set[j][1][0]
                                        ↪== 'diamond' and set[j][1][1][0]
                                        ↪== 'box':
                                        pass
                                    elif set[j][0] == 'box' and set[j][1][0]
                                        ↪ == 'diamond':
                                        pass
                                    elif set[j][0] == 'not' and set[j][1][0]
                                        ↪ == 'diamond':
                                        formula = ('not',set[j][1][1])
                                        if formula not in graph.node[
                                            ↪new_node]:
                                            graph.node[new_node].append(
                                                ↪formula)
                                            alpha_node_solve(graph, new_node
                                                ↪)
                                            beta_node_solve(graph, new_node,
                                                ↪ formulas_in)
                                    elif set[j][0] == 'box':
                                        if set[j][1] not in graph.node[
                                            ↪new_node]:
                                            graph.node[new_node].append(set[
                                                ↪j][1])
                                            alpha_node_solve(graph, new_node
                                                ↪)
                                            beta_node_solve(graph, new_node,
                                                ↪ formulas_in)
    '''
        :solving gammas at a NODE in graph
    '''
    def gamma_node_solve(graph, node, formulas_in):

        value_list = graph.node[node]
        size = len(graph.node[node])
        status = 1;
        index = 0;
        while status == 1:
            for i in range(index,size):
                value = value_list[i]
                if value[0] == 'box':
                    formula = value[1]
```

```
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if value[0] == 'box' and value[1][0] == "diamond":
                        value_list.remove(value)
                        graph.node[node]= value_list
                        if formula not in graph.node[node]:
                            graph.node[node].append(formula)
                    else:
                        for single_node in graph.nodes():
                            if formula not in graph.node[single_node]:
                                graph.node[single_node].append(formula)

            elif value[0] == 'not' and value[1][0] == "diamond":
                formula = ('not', value[1][1])
                if value not in formulas_in[node]:
                    formulas_in[node].append(value)
                    if value[0] == 'not' and value[1][0] == "diamond"
                        ↪and value[1][1][0] == 'box':
                        value_list.remove(value)
                        graph.node[node]= value_list
                        formula = ('not',value[1][1])
                        if formula not in graph.node[node]:
                            graph.node[node].append(formula)
                    else:
                        for single_node in graph.nodes():
                            if formula not in graph.node[single_node]:
                                graph.node[single_node].append(formula)
        new_size = len(graph.node[node])

        if size == new_size:
            status = 0;
        else:
            diff = new_size - size
            index = len(graph.node[node])-diff
            size = new_size

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""              Main loop iterating over graphs            """
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
def main():
    num_graph = 0
    for graph in Graphs:
        formulas_in = graph_formulas[num_graph]
        status = 1;
        index = 1;
        alpha_node(graph)
        while status == 1:
            for node in range(index,len(graph.nodes())+1):
                start_length = len(graph.nodes())

                alpha_node_solve(graph,node)
                beta_node_solve(graph, node,formulas_in)

                gamma_node_solve(graph, node, formulas_in)

                delta_node_solve(graph, node,formulas_in)

                alpha_node_solve(graph,node)

                beta_node_solve(graph, node,formulas_in)

                gamma_node_solve(graph, node, formulas_in)

                delta_node_solve(graph, node, formulas_in)

                end_length = len(graph.nodes())
                if start_length < end_length:
                    diff = end_length - start_length
                    index = index+1
                elif index < len(graph.nodes()):
                    index = index+1
                else:
                    status = 0;
        num_graph += 1

    '''
        :finding inconsistencies in the model
    '''
    index_inconsistent =[]
    for i in range(0,len(Graphs)):
        graph = Graphs[i]
        for node in graph.nodes():
            consistent_list = graph.node[node]
            status = sols.inconsistent(consistent_list)
            if status == True:
                index_inconsistent.append(i)
            else:
                status == False
    index_inconsistent = list(set(index_inconsistent))
    # removing inconsistent graphs- models
    if index_inconsistent is not []:
        for num in reversed(index_inconsistent):
            del Graphs[num];

    '''
        :display and save as pictures all the exiting graphs in the list
    '''
    gr.final_graphs(Graphs,psi)

t0 = time.clock()
main()
print ((time.clock() - t0), " seconds process time")
```

THE END