Notes for AI504 Knowledge Representation

Matteo Acclavio

February 25, 2025

These notes are a draft (Work In Progress) and are subject to change.
Please, do not distribute.
If you have any suggestion, please let me know.

Contents

1 A little bit of history

In a private letter from March 1706, Gottfried Wilhelm von Leibniz wrote about his works on '*characteristica universalis*', a formal language he imagined in which it would be possible to express mathematical, scientific, and metaphysical concepts.

It is true that in the past I planned a new way of calculating suitable for matters which have nothing in common with mathematics, and if this kind of

logic were put into practice, every reasoning, even probabilistic ones, would be like that of the mathematician: if need be, the lesser minds which had application and good will could, if not accompany the greatest minds, then at least follow them. For one could always say: let us calculate, and judge correctly through this, as much as the data and reason can provide us with the means for it. But I do not know if I will ever be in a position to carry out such a project, which requires more than one hand; and it even seems that mankind is still not mature enough to lay claim to the advantages which this method could provide.

Moved by a similar vision, at the International Congress of Mathematicians of 1900, David Hilbert proposed, among his 23 problems, the problem of checking coherence of mathematics. The problem was to find a formal system that could be used to prove all mathematical truths, and only mathematical truths. In 1931, Kurt Gödel showed that such a system could not exist, as it would be either incomplete or inconsistent. However, the idea of a formal system to represent knowledge, provided with a reasoning engine, was born.

With the advent of computers, the idea of a formal system to represent knowledge became more and more concrete. In 1956, John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon organized the Dartmouth Conference, where the term 'artificial intelligence' was coined. The goal of artificial intelligence was to create machines capable of performing tasks that required human intelligence, such as reasoning, learning, perception, and problem-solving.

Expert systems emerged in the mid-1960s, with applications in various fields, including medical applications. Two initial approaches to knowledge representation were developed:

- **General Problem Solver**: This approach used data structures to represent knowledge and a mechanism for decomposing tasks.
- **Advice Taker**: This approach used predicate calculus to model common sense reasoning.

Many works following the first approach represent knowledge by means of graph representations and semantic networks (similar to knowledge graphs). In such graphs, problem-solving becomes a form of graph traversal. At the same time, automated theorem provers were developed to reason in frameworks based on first-order logic, following the second approach.

The 'conflict' between those who wanted to consider knowledge as a set of static facts and those who wanted to consider knowledge as part of the inference mechanism itself was resolved with the introduction of ProLog. The early development of logic programming advocated the representation of domain-specific knowledge rather than general-purpose reasoning.

During the years leading up to the mid-1970s, the expectations for what expert systems could achieve in various fields were highly optimistic. Early research aimed to develop fully automated (i.e., entirely computerized) expert

systems. However, the expectations of what computers could accomplish were often overly idealistic. Karp's work [**?**] highlighted the limitations of expert systems due to the complexity of certain problems, making unrealistic to have 'efficient' way to provide solutions to common problems due to space and time constraints.

One of the first uses of ProLog was in the legal area. A notable use case was 'The British Nationality Act as a Logic Program' in 1986, which modeled the law from 1981 [**?**]. This work became a benchmark for law in AI. Another major boost to the development of expert systems was the release of the IBM PC in 1981, which led to the proliferation of new computer architectures, closed networks, and the client-server model.

2 Classical Propositional Logic

Propositional logic is a formal system in which the basic units of knowledge are propositions, which are statements that can be either true or false.

Formulas are sequences of symbols generated from a set $\mathcal{A}$ of **atomic proposition** using the following grammar:

$$\begin{array}{lccccccccc} A, B \coloneqq & p & | & \neg A & | & A \wedge B & | & A \vee B & | & A \to B \\ & \text{atom} & & \text{negation} & & \text{conjunction} & & \text{disjunction} & & \text{implication} \end{array}$$

We call **literal** any formula which is an atom or the negation of an atom.

The semantics of propositional logic is defined by **truth assignments**, which are functions that assign a truth value to each atomic proposition. Formally, a model for propositional logic is a function $\mathfrak{M} : \mathcal{F} \to \{\mathtt{ff}, \mathtt{tt}\}$ that assigns to each atomic proposition a truth value $\mathtt{ff}$ for false and $\mathtt{tt}$ for true. Equivalently, a model can be seen as a subset formulas $\mathcal{F}$ that contains all formulas which are evaluated as true.

The truth value of a formula is defined by fixing truth values for atomic propositions, that is by fixing a truth value $(a)^{\mathfrak{M}} \in \{\mathtt{tt}, \mathtt{ff}\}$ for each $a \in \mathcal{A}$, and by letting the truth value of a formula be determined by the truth values of its sub-formulas as follows:

$$\begin{aligned} (A \wedge B)^{\mathfrak{M}} &= \begin{cases} \mathtt{ff} & \text{if } (A)^{\mathfrak{M}} = \mathtt{ff} \text{ or } (B)^{\mathfrak{M}} = \mathtt{ff} \\ \mathtt{tt} & \text{otherwise} \end{cases} \\ (A \vee B)^{\mathfrak{M}} &= \begin{cases} \mathtt{ff} & \text{if } (A)^{\mathfrak{M}} = \mathtt{ff} \text{ and } (B)^{\mathfrak{M}} = \mathtt{ff} \\ \mathtt{tt} & \text{otherwise} \end{cases} \\ (A \to B)^{\mathfrak{M}} &= \begin{cases} \mathtt{ff} & \text{if } (A)^{\mathfrak{M}} = \mathtt{tt} \text{ and } (B)^{\mathfrak{M}} = \mathtt{ff} \\ \mathtt{ff} & \text{otherwise} \end{cases} \\ (\neg A)^{\mathfrak{M}} &= \begin{cases} \mathtt{ff} & \text{if } (A)^{\mathfrak{M}} = \mathtt{tt} \\ \mathtt{tt} & \text{if } (A)^{\mathfrak{M}} = \mathtt{ff} \end{cases} \end{aligned} \tag{1}$$

A	B	$A \wedge B$	$A \vee B$	$A \to B$	$\neg A$
$\mathfrak{ff}$	$\mathfrak{ff}$	$\mathfrak{ff}$	$\mathfrak{ff}$	$\mathfrak{tt}$	$\mathfrak{tt}$
$\mathfrak{ff}$	$\mathfrak{tt}$	$\mathfrak{ff}$	$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{tt}$
$\mathfrak{tt}$	$\mathfrak{ff}$	$\mathfrak{ff}$	$\mathfrak{tt}$	$\mathfrak{ff}$	$\mathfrak{ff}$
$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{ff}$

Figure 1: Truth table for the logical operators of propositional logic.

Remark 1. *In propositional logic, the truth value of a formula is determined by the truth values of the atoms in it, and it is computed by a recursive procedure on the structure of the formula. This procedure is the same as computing the* ***truth table*** *of the given formula – see Figure 1.*

A formula A is **satisfied** by a model $\mathfrak{M}$ (written $\mathfrak{M} \vDash A$) if A is evaluated as true in $\mathfrak{M}$. We say that a formula A **entails** a formula B (written $A \vDash B$) if every model that satisfies A also satisfies B, that is, if $\mathfrak{M} \vDash A$, then $\mathfrak{M} \vDash B$ for every model $\mathfrak{M}$. The process of checking whether a formula A entails a formula B is called **model checking**.

We say that a formula A is **satisfiable** if there exists a model $\mathfrak{M}$ such that $\mathfrak{M} \vDash A$. A formula A is **valid** if it is evaluated as true in every model, that is if $\mathfrak{M} \vDash A$ for every model $\mathfrak{M}$. A formula A is **unsatisfiable** if there is no model satisfying A. In this case we write $\nvDash A$.

By definition, we have the following theorem that relates entailment and implication, known as *deduction theorem*.

Theorem 2 (Deduction theorem)**.** *Let A and B formulas. Then, A entails B if and only if the implication $A \to B$ is valid. That is,*

$$A \vDash B \quad \textit{iff} \quad \vDash A \to B.$$

Proof. Let $\mathfrak{M}$ be a model. By definition of entailment, if $\mathfrak{M} \vDash A$, then $\mathfrak{M} \vDash B$. It follows that $\mathfrak{M}(A \to B) = \mathfrak{tt}$, that is, $\mathfrak{M} \vDash A \wedge B$.

Conversely, let $\mathfrak{M}$ be a model such that $\mathfrak{M} \vDash A \wedge B$. Then, by definition of the truth table of the implication, either $\mathfrak{M} \vDash A$ and $\mathfrak{M} \vDash B$, thus $\mathfrak{M} \vDash A \to B$, or $\mathfrak{M} \nvDash A$, and in this case the implication is evaluated as true. □

2.1 Logical Equivalence

Two formulas are **logically equivalent** if they are satisfied by the same models, that is, if $A \vDash B$ and $B \vDash A$.

Proposition 3. *Let A and B be formulas. Then, A and B are logically equivalent if and only if the formula $(A \to B) \wedge (B \to A)$ is valid.*

Proof. Let $\mathfrak{M}$ be a model. By definition of logical equivalence, A and B are logically equivalent if $\mathfrak{M} \vDash A$ if and only if $\mathfrak{M} \vDash B$. Using deduction theorem,

Algorithm 1: Negation Normal Form

Input: A formula A.
Output: A formula B in NNF equivalent to A.

return A if A is a literal;
return $\mathsf{NNF}(A_1) \odot \mathsf{NNF}(A_2)$ if $A = A_1 \odot A_2$ with $\odot \in \{\wedge, \vee, \rightarrow\}$;
return A_1 if $A = \neg\neg A_1$;
return $\mathsf{NNF}(\neg A_1) \vee \mathsf{NNF}(\neg A_2)$ if $A = \neg(A_1 \wedge A_2)$;
return $\mathsf{NNF}(\neg A_1) \wedge \mathsf{NNF}(\neg A_2)$ if $A = \neg(A_1 \vee A_2)$;
return $\mathsf{NNF}(A_1) \wedge \mathsf{NNF}(\neg A_2)$ if $A = \neg(A_1 \rightarrow A_2)$;

this implies that $\vDash A \rightarrow B$ and $\mathfrak{M} \vDash B \rightarrow A$, thus $\mathfrak{M} \vDash (A \rightarrow B) \wedge (B \rightarrow A)$. Conversely, if $\mathfrak{M} \vDash (A \rightarrow B) \wedge (B \rightarrow A)$, then $\mathfrak{M} \vDash A \rightarrow B$ and $\mathfrak{M} \vDash B \rightarrow A$. We conclude by deduction theorem that $A \vDash B$ and $B \vDash A$, that is the definition of logically equivalent. □

For some applications we discuss in the next sections, we are interested in some formulas in a specific form.

Definition 4. *Let A be a formulas. We say that:*

- *A is a **conjunctive clause** if it is a conjunction of literals;*
- *A is a **disjunctive clause** if it is a disjunction of literals;*
- *A is in **negation normal form** (NNF) if it is a formula in which negations are only applied to atoms;*
- *A is in **conjunctive normal form** (CNF) if it is a conjunction of disjunctive clauses.*
- *A is in **disjunctive normal form** (DNF) if it is a disjunction of conjunctive clauses.*

Given a formula A, we can always find a logically equivalent formula B in NNF by applying the Algorithm 1 and a logically equivalent formula C in CNF by applying the Algorithm 2.

Proposition 5. *Let A be a formula, $\mathsf{NNF}(A)$ be the formula returned by the Algorithm 1, and $\mathsf{CNF}(A)$ be the formula returned by the Algorithm 2. Then, A, $\mathsf{NNF}(A)$ and $\mathsf{CNF}(A)$ are logically equivalent.*

Proof. Exercise by induction on the structure of the formula A. □

3 Formal Reasoning in Propositional Logic

Because of the applications of propositional logic in automated reasoning, it is important to have (efficient) algorithms to check whether a formula B entails a

Algorithm 2: Conjunctive Normal Form

Input: A formula A.
Output: A formula C in CNF equivalent to A.

Let $B = \mathsf{NNF}(A)$;
if B *is a conjunctive or disjunctive clause* **then**
  return B;
end
if $B = B_1 \wedge B_2$ **then**
  return $\mathsf{CNF}(B_1) \wedge \mathsf{CNF}(B_2)$;
end
if $B = B_1 \vee (B_2 \wedge B_3)$ **then**
  return $\mathsf{CNF}(B_1 \vee B_2) \wedge \mathsf{CNF}(B_1 \vee B_3)$;
end

formula A. We say that an algorithm that checks whether a formula B entails a formula A is:

- **Sound** ($\vdash \Rightarrow \models$), that is, it always returns true when B entails A; and
- **Complete** ($\models \Rightarrow \vdash$), that is, if B entails A, then the algorithm returns true.

When designing an algorithm to check entailment, we aim to design an algorithm that is both sound and complete: soundness ensures that the algorithm is correct, that is, that it never validate an entitlement which is not true in some model, while completeness ensures that the algorithm is 'infallible' in its answers.

In this subsection we are interested in providing algorithms to answer the following problem:

Given a knowledge base KB, does KB entail A?

For this, we consider some algorithms to check entailment in propositional logic, where the knowledge base KB is encoded as (the conjunction of) a set of formulas. Given an algorithm EA to check entailment, we write $\mathsf{KB} \vdash_{\mathsf{EA}}$ to denote that EA is capable of deciding whether the formula encoding the KB entails A.

3.1 Model Checking with Brute Force

A simple algorithm to check entailment is based on truth tables, and it consists of checking if each model for the knowledge base KB satisfies the formula A.

The idea is to list all possible models for KB, that is, all the possible assignation of truth values to the atomic propositions such that KB is valid, and then check whether the formula A is evaluated as true in all these models. Thus, this method is not efficient, as it requires at least 2^n evaluations for each formula with n atomic propositions in KB and in A, making it impractical for applications which may consider large formulas.

Algorithm 3: Model Checking Algorithm

Input: A knowledge base KB and a formula A.
Output: True if KB entails A, false otherwise.

```
for each model 𝔐 such that 𝔐 ⊨ KB do
    if 𝔐 ⊭ A then
        return false;
    end
end
return true;
```

Remark 6. *We could decide to limit our attention to the 'minimal' models in which we define the truth value of the atoms occurring in A. In fact, thanks to the deduction theorem (Theorem 2), we know that $KB \models A$ iff $\models KB \to A$. Therefore, we could limit our attention to the models in which A is evaluated as $\mathtt{t}$: the only possibility for $KB \to A$ to be evaluated as $\mathtt{f}$ is when KB is evaluated as $\mathtt{t}$ and A as $\mathtt{f}$, while if KB is evaluated as $\mathtt{f}$, then $KB \to A$ is evaluated as $\mathtt{t}$ no matter of the value of A.*

We write $\mathsf{KB} \vdash_{\mathsf{MC}} A$ is the model checking algorithm from Algorithm 3, after constructing all the (relevant) models for KB, return true only if every such model for KB is also a model for A. We have the following theorem.

Theorem 7. *The algorithm* MC *is sound and complete with respect to entailment. That is,*

$$KB \vdash_{\mathsf{MC}} A \quad \textit{if and only if} \quad KB \models A.$$

Proof. Let KB be a knowledge base, and A a formula.

- Soundness: if $\mathsf{KB} \vdash_{\mathsf{MC}} A$, then for any model $\mathfrak{M}$ such that $\mathfrak{M} \models \mathsf{KB}$ we also have that $\mathfrak{M} \models A$. That is, $\mathsf{KB} \models A$.
- Completeness: $\mathsf{KB} \models A$, then for any model $\mathfrak{M}$ such that $\mathfrak{M} \models \mathsf{KB}$ we also have that $\mathfrak{M} \models A$. By definition, this means that $\mathsf{KB} \vdash_{\mathsf{MC}} A$. □

3.2 A Simple inference system for entailment

We can define a simple inference system for entailment in propositional logic, which is based on the following rule, called **Modus Ponens**, stating that if A and $A \to B$ are valid, then we can conclude that also B is valid.

$$\mathsf{MP}\ \frac{A \quad A \to B}{B} \tag{2}$$

We can easily check that the Modus Ponens rule is sound, that is, if both its premises are valid, then its conclusion is valid.

Lemma 8. *The Modus Ponens rule is sound with respect to entailment. That is, if A and $A \to B$ are valid, then B is valid. That is,*

$$\vDash A \wedge (A \to B) \to B.$$

Proof. Let $\mathfrak{M}$ be a model. By definition of the truth table of the implication, if $\mathfrak{M} \vDash A$ and $\mathfrak{M} \vDash A \to B$, then $\mathfrak{M} \vDash B$. □

Remark 9. *Another rule that could be useful to check entailment is the **And-Elimination** rule, which states that if $A \wedge B$ is valid, then both A (and B) is valid. This rule is useful for extracting from some formulas from a knowledge base if this latter is represented as the conjunction of formulas. However, if we consider knowledge bases as sets of formulas rather than their cojunction, then this rule is not necessary.*

$$\wedge\mathsf{E}\,\frac{A \wedge B}{A} \quad \textit{and} \quad \wedge\mathsf{E}\,\frac{A \wedge B}{A} \tag{3}$$

Prove sondness for this rule is immediate because a model satisfies $A \wedge B$ iff it satisfies both A and B.

The Modus Ponens is the unique rule we need to infer properties from a knowledge base espressed in classical propositional logic. Note that, because of the semantics of classical logic, the following axioms have to be considered in order to capture in a purely syntactical way the entailment relation.

$$\begin{array}{ll}
\text{(A1)} & A \to (B \to A) \\
\text{(A2)} & (A \to (B \to C)) \to ((A \to B) \to (A \to C)) \\
\text{(A3)} & A \wedge B \to A \\
\text{(A4)} & A \wedge B \to B \\
\text{(A5)} & A \to (B \to (A \wedge B)) \\
\text{(A6)} & A \to A \vee B \\
\text{(A7)} & B \to A \vee B \\
\text{(A8)} & (A \to C) \to ((B \to C) \to (A \vee B \to C)) \\
\text{(A9)} & (A \to B) \to ((A \to \neg B) \to \neg A) \\
\text{(A10)} & \neg\neg A \to A
\end{array} \tag{4}$$

These axioms are required solely to capture the following logical equivalence between formulas dictated by the truth tables of the logical operators.

Formula	**Name**
$(A \wedge B) \equiv (B \wedge A)$	Commutativity of $\wedge$
$(A \vee B) \equiv (B \vee A)$	Commutativity of $\vee$
$((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))$	Associativity of $\wedge$
$((A \vee B) \vee C) \equiv (A \vee (B \vee C))$	Associativity of $\vee$
$\neg(\neg A) \equiv A$	Double-negation elimination
$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$	De Morgan law for conjunction
$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$	De Morgan law for disjunction
$(A \wedge (B \vee C)) \equiv ((A \wedge B) \vee (A \wedge C))$	Distributivity of $\wedge$ over $\vee$
$(A \vee (B \wedge C)) \equiv ((A \vee B) \wedge (A \vee C))$	Distributivity of $\vee$ over $\wedge$

(5)

For example, the axiom (A3) and (A4) corresponds to the definition of the conjunction, while the axiom (A6) and (A7) corresponds to the definition of the disjunction.

3.3 Checking Entailment via Resolution

As a consequence of the deduction theorem (Theorem 2), we have that the entailment can be checked using unsatisfiability, which we are going to use in the next

Corollary 10. *Let A and B be formulas. Then, A entails B iff $A \wedge \neg B$ is unsatisfiable. That is,*

$$A \models B \quad \textit{if and only if} \quad \not\models A \wedge \neg B \,. \tag{6}$$

Proof. By the deduction theorem we know that $A \models B$ iff $\models A \to B$. This is equivalent to say that $A \to B$ must be evaluated as $\mathtt{tt}$ in all possible models.

We now remark that $A \to B$ is evaluated as $\mathtt{tt}$ in a model $\mathfrak{M}$, iff $\neg A \vee B$ is evaluated as $\mathtt{tt}$, iff $\neg(\neg A \vee B)$ is evaluated as $\mathtt{ff}$, and iff $A \wedge \neg B$ is evaluated as $\mathtt{ff}$.

Then, if $A \to B$ is evaluated as $\mathtt{tt}$ in all models, then $A \wedge \neg B$ must be evaluated as $\mathtt{ff}$ in all models. This means that, in particular, $A \wedge \neg B$ is unsatisfiable. □

Said differently, a formula A is entailed by a knowledge base KB if and only if there is no model $\mathfrak{M}$ such that $\mathfrak{M} \models \mathsf{KB}$ and $\mathfrak{M} \not\models A$. This observation suggests a simple algorithm to check entailment, which is based on the following idea: if you can find a model $\mathfrak{M}$ such that $\mathfrak{M} \models \mathsf{KB}$ and $\mathfrak{M} \not\models A$, then KB does not entail A; otherwise KB entails A.

The algorithm we use to check entailment is based on the resolution rule. First, to make the syntax more readable, we represent conjunctive normal form formulas as lists of clauses. That is, we simply write

$$[\ell_{1,1}, \ldots, \ell_{1,n_1}], \ldots, [\ell_{m,1}, \ldots, \ell_{m,n_m}]$$

to denote the CNF formula

$$(\ell_{1,1} \vee \cdots \vee \ell_{1,n_1}) \wedge \cdots \wedge (\ell_{m,1} \vee \cdots \vee \ell_{m,n_m}) \,.$$

The **empty clause** [] has to be interpreted as the false, or any unsatisfiable formula. From now on, we identify a CNF-formula A with the set of its clauses, that is, we can write either $A = \{C_1, \ldots, C_n\}$ or $A = C_1 \wedge \cdots \wedge C_n$.

Remark 11. *Both conjunction and disjunction are* idempotent *in classical logic, that is, $A \wedge A \equiv A$ and $A \vee A \equiv A$. This remark allows us to consider a CNF formula as a* set *of clauses, and clauses as* sets *of literals. This avoids us to have to deal with repetitions of literals in the clauses, and repetitions of clauses in the CNF formula.*

With this notation, we can write the resolution rule as follows, where ℓ and ℓ' are two dual literals.

$$\mathsf{Res}\ \frac{[\ell_1, \ldots, \ell_{k-1}, \ell], [\ell', \ell_k, \ldots, \ell_m]}{[\ell_1, \ldots, \ell_n]} \tag{7}$$

The conclusion of the rule is called **resolvant**. For example, the resolution rule applied to the clauses $[\neg s, p, q, r]$ and $[\neg p, t]$ gives the resolvent clause $[\neg s, q, r, t]$. We call a **resolution derivation** a sequence of applications of the resolution rule, starting from a set of clauses A and ending with the empty clause $[]$.

Notation 12. *When representing resolution derivations we may just write the sequence of clauses that are used by the resolution rule. That is, we may write the resolution derivation below on the left instead of writing the one on the right (which keeps a copy of all derived clauses).*

$$\mathsf{Res}\ \frac{\mathsf{Factoring}\ \frac{\mathsf{Res}\ \frac{\mathsf{Res}\ \frac{[s,p],[\neg s,p,q],[\neg q],[\neg p]}{[s,p],[\neg s,p],[\neg p]}}{[p,p],[\neg p]}}{[p],[\neg p]}}{[]} \qquad \mathsf{Res}\ \frac{\mathsf{Factoring}\ \frac{\mathsf{Res}\ \frac{\mathsf{Res}\ \frac{[s,p],[\neg s,p,q],[\neg q],[\neg p]}{[s,p],[\neg s,p],[\neg p],[\neg s,p,q],[\neg q]}}{[p,p],[\neg p],[s,p],[\neg s,p],[\neg p],[\neg s,p,q],[\neg q]}}{[p],[\neg p],[s,p],[\neg s,p],[\neg p],[\neg s,p,q],[\neg q]}}{[],[p],[\neg p],[s,p],[\neg s,p],[\neg p],[\neg s,p,q],[\neg q]}$$

In some sense, the derivation on the left is the 'resource aware' version of the derivation on the right, where we only keep track of the clauses that are used by the instances of resolution rules.

Note that, when applying the resolution rule, we do not necessary have to remove the two clauses that are used in the resolution, but we can keep them in the set of clauses. In addition, depending on which mathematical structure we are assuming for the clauses (i.e., sets or lists), we may need to consider the so-called **factoring rule**, which states that if a clause contains two literals that are the same, then one of the two literals can be removed.

We have the following result:

Lemma 13. *The resolution rule is sound. That is, if the premise of the rule is a valid set of clauses, then the conclusion also is.*

Proof. We observe that if a model $\mathfrak{M}$ satisfies a set of clauses must satisfy all of them. In particular, it satisfies the two clauses in the premises of the resolution rule. Without loss of generality, we start our reasoning on the left-hand side clause in the premise of the rule, but the same reasoning can be applied by starting from the right-hand side clause.

Since $\mathfrak{M} \vDash [\ell_1, \ldots, \ell_{k-1}, \ell]$, then $\mathfrak{M} \vDash \ell_1 \vee \cdots \vee \ell_{k-1}$ or $\mathfrak{M} \vDash \ell$. In the first case, $(\ell_1 \vee \cdots \vee \ell_{k-1})^{\mathfrak{M}} = \mathtt{tt}$ and we have can conclude that $\mathfrak{M} \vDash \ell_1 \vee \cdots \vee \ell_n$ because $(A \vee B)^{\mathfrak{M}} = \mathtt{tt}$ whenever $(A)^{\mathfrak{M}} = \mathtt{tt}$ and $\ell_1 \vee \cdots \vee \ell_n = (\ell_1 \vee \cdots \vee \ell_{k-1}) \vee (\ell_k \vee \cdots \vee \ell_n)$. Otherwise, we have that $(\ell)^{\mathfrak{M}} = \mathtt{tt}$, thus $(\ell')^{\mathfrak{M}} = \mathtt{ff}$ because ℓ and ℓ' are dual literals. Since $(\ell', \ell_k, \ldots, \ell_m)^{\mathfrak{M}} = \mathtt{tt}$, then we must have $(\ell_k \vee \cdots \vee \ell_m)^{\mathfrak{M}} = \mathtt{tt}$. We

then conclude that $\mathfrak{M} \vDash [\ell_1, \ldots, \ell_{k-1}, \ell', \ell_k, \ldots, \ell_m]$ for the same argument on the evaluation of the disjunction as above. □

We want to show that we can use resolution to check validity of a CNF-formula. To do so, we define the **resolution closure** of a set of clauses A as the set of all clauses that can be derived from A recursively using the resolution rule. We denote the resolution closure of A as $\overline{\mathsf{Res}}(A)$.

Lemma 14. *If C and C' are sets of clauses, with $C \subseteq C'$, then $\overline{Res}(C) \subseteq \overline{Res}(C')$.*

Proof. Left as an exercise. □

Remark 15. *In propositional logic, it is always possible to compute the resolution closure of a finite set of clauses. This because the set of possible clauses derivable from a set of clauses is finite. More precisely, the set of clauses over n atoms contains 4^n elements: each clause can contain the atom, its negation, or none of them, or both of them.*

Theorem 16. *Let A be a set of clauses (i.e., a CNF-formula). Then A is valid iff there is no resolution derivation of $[]$ from A.*

Proof. Lemma 13 tells us that the resolution rule is sound, therefore if A is valid, then any set of clauses that can be derived from A using the resolution rule must be valid. The set of clauses containing the empty clause $[]$ is unsatisfiable, therefore it cannot be derived using resolution rule from a valid set of clauses.

We prove the converse by contraposition. That is, we show that if we can derive the empty clause $[]$ from A using the resolution rule, then A must be unsatisfiable. We reason by induction on the number of atoms occurring in A:

- if A is a CNF-formula containing only one atom, and we can derive the empty clause $[]$ from A using the resolution rule, then $A = [a][\neg a]$[1] $\left(\text{because } \mathsf{Res}\,\dfrac{[a][\neg a]}{[]}\right)$;
- if A contains $n > 1$ atoms, then we can write A as

$$A = \underbrace{C_1, \ldots C_k}_{\text{clauses containing } a} \quad , \quad \underbrace{C_{k+1}, \ldots C_m}_{\text{clauses not containing } a \text{ or } \neg a} \quad , \quad \underbrace{C_{m+1}, \ldots C_n}_{\text{clauses containing } \neg a}$$

  If we can derive the empty clause $[]$ from A using the resolution rule, and $[]$ is in the resolution closure of $C_{k_1}, \ldots C_m$, then we can conclude that A is unsatisfiable. In fact, $A' = C_{k_1}, \ldots C_m$ is a set of clauses containing strictly less than n atoms, and we can apply the inductive hypothesis to conclude that A' is unsatisfiable. Thus also $A = A' \wedge B$ is unsatisfiable.

  Otherwise, we must have that $[]$ has been derived by applying the resolution rule to two clauses $[a]$ and $[\neg a]$, which should belong to $\overline{\mathsf{Res}}(A)$.

[1] More in general, we should say that A must be of the form $[a, \ldots, a][\neg a, \ldots, \neg a]$.

Thus, we should have two resolution derivations of $[a]$ and $[\neg a]$ of the following form:

$$\begin{array}{c|c} C_1, \ldots, C_k, C_{k+1}, \ldots, C_m & C_{k+1}, \ldots, C_m, C_{m+1}, \ldots, C_n \\ \| & \| \\ [a] & [\neg a] \end{array}$$

Note that in the derivation on the left-hand side, no clause in $\{C_{m+1}, \ldots, C_m\}$ is considered, and in the derivation on the right-hand side, no clause in $\{C_1, \ldots, C_k\}$ is considered. This means that we can construct two resolution derivations of [] from a set of clauses $B = A \setminus \{a, \neg a\}$ obtained by removing a and $\neg a$ from A.

$$\begin{array}{c|c} C_1 \setminus \{a\}, \ldots, C_k \setminus \{a\}, C_{k+1}, \ldots, C_m & C_{k+1}, \ldots, C_m, C_{m+1} \setminus \{\neg a\}, \ldots, C_n \setminus \{\neg a\} \\ \| & \| \\ [] & [] \end{array}$$

By the inductive hypothesis, we can conclude that both sets of clauses $B_1 = \{C_1, \ldots, C_k, C_{k+1}, \ldots, C_m\} \setminus \{a\}$ and $B_1 = \{C_1, \ldots, C_k, C_{k+1}, \ldots, C_m\} \setminus \{\neg a\}$ are unsatisfiable. From this, we conclude that A must be unsatisfiable: if A would be satisfiable, then there exists a model $\mathfrak{M}$ such that $\mathfrak{M} \vDash A$. In such a model $\mathfrak{M}$ we would have that

- either $(a)^{\mathfrak{M}} = \mathtt{tt}$ and $\mathfrak{M} \vDash C_{k+1} \wedge \cdots \wedge C_m \wedge C_{m+1} \wedge \cdots \wedge C_m$.
- or $(\neg a)^{\mathfrak{M}} = \mathtt{tt}$ and $\mathfrak{M} \vDash C_1 \wedge \cdots \wedge C_k \wedge C_{k+1} \wedge \cdots \wedge C_m$;

In the first case, we would have that $\mathfrak{M} \vDash C_i$ for all $i \in \{k+1, \ldots, n\}$, and, since $(a)^{\mathfrak{M}} = \mathtt{tt}$ and $(\neg a)^{\mathfrak{M}} = \mathtt{ff}$, we would have that $\mathfrak{M} \vDash C_{m+1} \setminus \{\neg a\}, \ldots, C_n \setminus \{\neg a\}$. This implies that $\mathfrak{M} \vDash B_2$, contraddicting the fact that B_2 is unsatisfiable. In the second case, we can reason in the same way, reaching a contradiction with the fact that B_1 is unsatisfiable. Therefore, A must be unsatisfiable.

□

We can now define an algorithm to check entailment based on the resolution rule based on Corollary 10 defined as in Algorithm 4.

Remark 17. *We could consider additional rules to simplify the resolution process. For example:*

- *the **tautology rule**, which states that if a clause contains a literal and its negation, then the clause is a tautology and can be removed. Intuitively, since the clause is the disjunction of literals, if it contains a literal and its negation, then the clause is always evaluated as true. Therefore, it does not provide any information;*
- *the **subsumption rule**, which states that if a clause is a subset of another clause, then the clause can be removed. Here the intuition is that if a clause is a subset of another clause, then the superset clause already contains all the information provided by the subset clause.*

Algorithm 4: Resolution Algorithm

Input: A knowledge base KB and a formula A.
Output: True if KB entails A, false otherwise.

Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of clauses in CNF encoding $\mathsf{KB} \wedge \neg A$;
do
  Let $\mathcal{C}' = \emptyset$;
  for $i, j \in \{1, \ldots n\}$ *with* $i \neq j$ **do**
    Compute the resolvent C_{ij} of C_i and C_j (if possible);
    if $C_{ij} = []$ **then**
      return true;
    else
      $\mathcal{C}' = \mathcal{C}' \cup \{C_{ij}\}$;
    end
  end
while $\mathcal{C} \neq \mathcal{C} \cup \mathcal{C}'$;
return false;

Similar remarks can be used to improve the efficiency of the resolution algorithm. In particular, the ***David-Putnam*** *algorithm is a well-known algorithm that uses these (and other) rules to simplify the resolution process.*

We write $\mathsf{KB} \vdash_{\mathsf{Res}} A$ it is possible to derive the empty clause [] from the set of clauses $\mathsf{KB} \wedge \neg A$ using the resolution rule. We have the following result.

Theorem 18. *The algorithm* Res *is sound and complete with respect to entailment. That is,*

$$KB \vdash_{Res} A \quad \textit{if and only if} \quad KB \models A.$$

Proof. By Corollary 10 we know that $\mathsf{KB} \models A$ iff $\not\models \mathsf{KB} \wedge \neg A$. We conclude since we have proven in Theorem 16 that $\not\models \mathsf{KB} \wedge \neg A$ iff $\mathsf{KB} \vdash_{\mathsf{Res}} A$. □

3.4 Forward and Backward Chaining

We now discuss two approaches to infer properties from a knowledge base: the **forward chaining** and the **backward chaining**. The forward chaining is based on the idea of applying rules from known facts from the database to infer new facts. The backward chaining is based on the idea of applying rules from the goal to infer the facts that are needed to reach the goal.

In both cases, we consider a knowledge base KB containing only clauses of a specific shape called **definite clause**, which are (disjunctive) clause containing at most one positive literal. The **head** of a definite clause is the positive literal, and the **body** is the conjunction of the negative literals. Among definite clauses, we distinguish **facts**, which are clauses made of a single positive literal, and **Horn clauses**, which are clauses containing exactly one positive literal. The interest in consider definite clauses, is that each definite clause can be seen as

an implication with antecedent the conjunction of the negative literals and the consequent the positive literal.

$$[\neg b_1, \ldots, \neg b_n, h] \stackrel{\text{def}}{=} (\neg b_1) \vee \cdots (\vee \neg b_n) \vee h \equiv (b_1 \wedge \cdots \wedge b_n) \rightarrow h$$

Alternatively, each Horn clause can be seen as a rule of inference, with conclusion the positive literal and premise(s) the negative literals, and each fact can be seen a rule with no premises.

$$[\neg b_1, \ldots, \neg b_n, h] \rightsquigarrow \frac{b_1 \quad \cdots \quad b_n}{h} \quad \text{and} \quad [h] \rightsquigarrow \frac{}{h}$$

With this interpretation of definite clauses in mind, checking if a query q is satisfied by a knowledge base can be seen as the process of constructing a tree with root q and leaves the facts in the knowledge base. Forward chaining constructs a forest of trees top-down (from leaves to the root), until a tree with root q is found. Backward chaining constructs a single tree bottom-up (from the root to the leaves), until a tree with leaves the facts in the knowledge base is found.

The **forward chaining** algorithm is defined as in Algorithm 5. The algorithm returns true if the query q can be inferred from the knowledge base KB, and false otherwise. The algorithm starts from the set $\mathcal{G}$ of facts (i.e., definite clauses which are facts) in the knowledge base KB, and it progressively add the facts which can be obtained by applying the rules in KB. For optimizing this procedure, the algorithm keeps track of the number of literals in the body of each clause, and it adds the head of the clause to the set of facts only when all the literals in the body have been added to the set of facts.

Example 19. *Consider the run of the forward chaining algorithm to check the entailment of the query Q on the following knowledge base:*

$$\begin{array}{lll} C_1 \coloneqq A & C_2 \coloneqq B & C_3 \coloneqq (A \wedge B) \rightarrow L \\ C_4 \coloneqq (A \wedge P) \rightarrow L & C_5 \coloneqq (B \wedge L) \rightarrow M & C_6 \coloneqq (L \wedge M) \rightarrow P \\ C_7 \coloneqq P \rightarrow Q & & \end{array}$$

When the algorithm starts, it sets $\mathcal{F} = \{A, B\}$, and it sets

$$n_{C_1} = 1,\ n_{C_2} = 1,\ n_{C_3} = 2,\ n_{C_4} = 2,\ n_{C_5} = 2,\ n_{C_6} = 2,\ n_{C_7} = 1.$$

Then the while loop starts, and the algorithm picks A from $\mathcal{F}$. Since A is not Q, the algorithm checks which clauses contain A in the body, that is, C_1, C_3 and C_4, letting $n_{C_1} = 0$, $n_{C_3} = 1$, and $n_{C_4} = 1$. Since $n_{C_1} = 0$, the algorithm adds the head of C_1, that is A, to the set of facts $\mathcal{F}$. Then it removes A from $\mathcal{F}$. Then the algorithm picks B from $\mathcal{F}$, and, since $B \neq Q$, it checks the clauses containing B in the body, that is, C_2, C_3 and C_5, letting $n_{C_2} = 0$, $n_{C_3} = 0$, and $n_{C_5} = 1$. Since $n_{C_2} = 0$ and $n_{C_3} = 0$, the algorithm tries to add the head of C_2, that is B, and the head of C_3, that is L, to the set of facts $\mathcal{F}$. However, since $[B]$ is a clause in the knowledge base, the algorithm only adds L to $\mathcal{F}$ (and $[L]$

Algorithm 5: Forward Chaining

Input: A knowledge base KB, and a query q.
Output: True if KB entails q, false otherwise.

```
Let F = {f | exists C ∈ KB is a fact [f]};
Let n_C be the number of literals in the body of each clause C ∈ KB;
while F ≠ ∅ do
    Pick an element f from F;
    if f = q then
        return true;
    else
        for C ∈ C do
            if f is in the body of C = [¬b_1, ..., ¬b_n, h] then
                n_C = n_C − 1;
                if n_C = 0 and [f] ∉ KB then
                    F = F ∪ {h} and KB = KB ∪ {[f]};
                end
            end
        end
        F = F \ {f};
    end
end
return false;
```

to KB). Then it removes B from $\mathcal{F}$. Then the algorithm continues by picking L and adding M, then picking M and adding P. At this point, we see that the algorithm will not add back L in $\mathcal{F}$, because the clause $[L]$ is already present in KB. Finally, the algorithm picks P and adds Q to the set of facts $\mathcal{F}$. Since Q is the query, the algorithm returns true.

We write $\mathsf{KB} \vdash_{\mathsf{FC}} q$ if the forward chaining algorithm returns true when applied to the knowledge base KB and the query q. We have the following result.

Theorem 20. *The forward chaining algorithm is sound and complete with respect to entailment. That is,*

$$KB \vdash_{FC} q \quad \text{if and only if} \quad KB \vDash q.$$

Proof. Soundness of the algorithm follows from the fact that the algorithm is simply applying the Modus Ponens rule – see Lemma 8.

To prove completeness, we consider a model $\mathfrak{M}$ such that $(f)^{\mathfrak{M}} = \mathtt{tt}$ for any f which ever occurs in the set of facts $\mathcal{F}$ during the execution of the forward chaining algorithm. For any such model, $\mathfrak{M} \vDash \mathsf{KB}$:

- if f is a fact in the knowledge base, that is, a fact clause, then $\mathfrak{M} \vDash f$ because f is in the initial set of facts $\mathcal{F}$;

- if f is a fact that has been added to the set of facts $\mathcal{F}$, then there must be a clause $C = [\neg b_1, \ldots, \neg b_n, h]$ in the knowledge base such that $f = h$ is the head of C and all atoms $b_1, \ldots, b_n$ in the body of C occurred in the set of facts $\mathcal{F}$. Since f is added to the set of facts $\mathcal{F}$, only when all the literals in the body of C occurred in $\mathcal{F}$. Then by the induction hypothesis, we must have that $(b_i)^{\mathfrak{M}} = \mathtt{tt}$ for each literal in the body, and $(f)^{\mathfrak{M}} = \mathtt{tt}$.

- if at least an atom b occurring in the body of C have not been added to the set of facts $\mathcal{F}$ during the execution of the forward chaining algorithm, then $(b)^{\mathfrak{M}} = \mathtt{ff}$ and $(f)^{\mathfrak{M}} = \mathtt{ff}$.

In all cases, every definite clause of the knowledge base is satisfied by the model $\mathfrak{M}$, and we conclude that $\mathfrak{M} \models \mathsf{KB}$. Assume now that q is valid in $\mathfrak{M}$, but q has never occurred in the set of facts $\mathcal{F}$ at the end of the execution of the forward chaining algorithm. For this to be true, there must be a clause $C = [\neg b_1, \ldots, \neg b_n, q]$ in the knowledge base such that $(b_i)^{\mathfrak{M}} = \mathtt{tt}$ for all i, and $(q)^{\mathfrak{M}} = \mathtt{ff}$. Now, since we have that $(b_i)^{\mathfrak{M}} = \mathtt{tt}$ for all i, then each b_i occurred in $\mathcal{F}$ at a certain point during the run of the algorithm. This means that n_C has become 0 at a certain point, thus that q has been added to the set of facts $\mathcal{F}$. This is a contradiction with the assumption we have made that $(q)^{\mathfrak{M}} = \mathtt{ff}$ because q has never occurred in the set of facts $\mathcal{F}$. Therefore, if q is valid in $\mathfrak{M}$, then q must have occurred in the set of facts $\mathcal{F}$ at the end of the execution of the forward chaining algorithm. □

The **backward chaining** algorithm is defined as in Algorithm 6. The algorithm returns true if the query q can be inferred from the knowledge base KB, and false otherwise. The algorithm starts from the query q and it applies 'backward' rules in KB to check under which pre-conditions the query q can be inferred, and repeats the process until the pre-conditions are facts in the knowledge base.

Note that, as written, it is easy to find simple knowledge bases and queries for which the backward chaining algorithm does not terminate, even if the answer is positive. For an example, consider the knowledge base $\mathsf{KB} = \{C_1 = [\neg p, q], C_2 = [\neg q, p], C_3 = [q]\}$ and the query q. The algorithm could enter into a loop by calling itself recursively on the set of queries $\{p\}$ (by applying the rule in C_2), then on the set of queries $\{q\}$ (by applying the rule in C_1), while it could have reached the goal by immediately applying the rule in C_3.

Example 21. *Consider a run of the backward chaining algorithm on the same query Q and database KB from Example 19.*

In this case, the algorithm starts from the query Q and it applies the rules in the knowledge base to infer the facts that are needed to reach the fact Q. The only possible rule is $C_7 \coloneqq P \to Q$. Therefore, the algorithm removes Q from the set of queries and adds P to $\mathcal{Q}$. Then it can only apply the rule $C_6 \coloneqq (L \wedge M) \to P$, removing P and adding L and M to $\mathcal{Q}$. Here the execution can go wrong: if the algorithm applies the rule C_4, it would remove L from $\mathcal{Q}$, but addign A, and adding back P. In this case, the algorithm would enter into a loop of applications

Algorithm 6: Backward Chaining

Input: A knowledge base KB, and a set of queries $Q = \{q\}$.
Output: True if KB entails q, false otherwise.

```
for each q ∈ Q do
    if there is C = [¬b_1, ..., ¬b_n, q] ∈ KB then
        Let Q = Q \ {q} ∪ {b_1, ..., b_n};
        if Q is empty then
            return true;
        else
            call Backward Chaining Algorithm on KB and Q;
        end
    else
        return false;
    end
end
```

of C_6 followed by C_4. However, if the algorithm applies the rule C_5, it would remove M from Q and add B and L. Then it could either apply C_2 to remove B, but in any case it should apply C_3 to remove L and add both A and B (or end again in the loop of C_6 and C_4). In this case, the algorithm would remove both A and B from Q via C_1 and C_2, exiting the loop of recursive calls and returning true since the set of queries is empty.

4 Propositional Modal Logic

Modal logic is an extension of propositional logic that includes modal operators allowing to express attributes of propositions, such as necessity, possibility, knowledge, belief, and time as the ones in the following table.

Logic	Symbols	Expressions Symbolized
Modal Logic	$\Box$	It is necessary that ...
	$\Diamond$	It is possible that ...
Deontic Logic	O	It is obligatory that ...
	P	It is permitted that ...
	F	It is forbidden that ...
Temporal Logic	G	It will always be the case that ...
	F	It will be the case that ...
	H	It has always been the case that ...
	P	It was the case that ...
Doxastic Logic	B_x	x believes that ...
Epistemic Logic	K_x	x knows that ...

We will focus on what we refer to as **modal logic**, which is a formal system that extends propositional logic with modal operators $\Box$ and $\Diamond$. The syntax of

formulas is defined by extending the syntex of propositional logic with modalities as follows:

$$A, B \coloneqq p \mid \neg A \mid A \wedge B \mid A \vee B \mid A \to B \mid \Diamond A \mid \Box A \quad p \in \mathcal{A}$$

4.1 Semantics of Modal Logic

The semantics of modal logic is defined by **Kripke models**. Intuitively, while models for propositional logic are structures made of a single *world* in which each proposition is evaluated as true or false, models for modal logic are made of different *possible worlds*, each provided with an evaluation function, and a relation between worlds. The relation between worlds is used to express the notion of accessibility between worlds, and it is used to define the semantics of the modal operators.

Formally, a (Kripke) model is a triple $\mathfrak{M} = (W, R, V)$ made of:

- a set of worlds W;
- a binary **accessibility** relation $R \subseteq W \times W$;
- a valuation function $V : W \times \mathcal{F} \to \{\mathtt{tt}, \mathtt{ff}\}$ that assigns to each world the set of propositions which are evaluated as true in that world.

We may write vRw to denote that the world w is accessible from the world v via R.

We write $\mathfrak{M}, w \vDash A$ to denote that the formula A is evaluated as true in the world w of the model $\mathfrak{M}$. The truth value of a formula A in a world w of a model $\mathfrak{M}$ is defined by induction on the structure of the formula as follows, under the assumption that the valuation function V is defined for each world w and for each $a \in \mathcal{A}$.

$$\begin{array}{ll}
\mathfrak{M}, w \vDash a & \text{if } V(w, a) = \mathtt{tt} \\
\mathfrak{M}, w \vDash \neg A & \text{if } \mathfrak{M}, w \nvDash A \\
\mathfrak{M}, w \vDash A \wedge B & \text{if } \mathfrak{M}, w \vDash A \text{ and } \mathfrak{M}, w \vDash B \\
\mathfrak{M}, w \vDash A \vee B & \text{if } \mathfrak{M}, w \vDash A \text{ or } \mathfrak{M}, w \vDash B \\
\mathfrak{M}, w \vDash A \to B & \text{if } \mathfrak{M}, w \nvDash A \text{ or } \mathfrak{M}, w \vDash B \\
\mathfrak{M}, w \vDash \Diamond A & \text{if } \exists v \in W \text{ such that } wRv \text{ and } \mathfrak{M}, v \vDash A \\
\mathfrak{M}, w \vDash \Box A & \text{if } \forall v \in W \text{ such that } wRv \text{ we have that } \mathfrak{M}, v \vDash A
\end{array} \tag{8}$$

The **frame** of the model is the graph (W, R) with vertices the set of worlds of the model, and with edges the accessibility relation. We can define the **frame condition** for a model $\mathfrak{M}$ as the condition that the accessibility relation R has, as, e.g., being *reflexive*, *transitive*, or *symmetric*.

We use the following notation to denote the entailment relation in modal logic, where $\mathfrak{M}$ is a model, w is a world, and A is a formula.

- $\mathfrak{M}, w \vDash A$ if A is evaluated as true in the world w of the model $\mathfrak{M}$;
- $\mathfrak{M} \vDash A$ if A is evaluated as true in all worlds of the model $\mathfrak{M}$;

- $\vDash A$ if A is evaluated as true in all possible models;

Moreover, for specific applications we may be interested in considering a class of models with underlying frame $\mathfrak{F}$ or with underlying frame in the class of frames $\mathfrak{C}$. In this case, we write:

- $\vDash_{\mathfrak{F}} A$ if A is evaluated as true in all models with frame $\mathfrak{F}$;
- $\vDash_{\mathfrak{C}} A$ if A is evaluated as true in all models whose frame is in the class $\mathfrak{C}$;

We can characterize frames with specific properties by defining *modal axioms*. That is, we can define a modal axiom $\mathsf{AX}_{\mathfrak{C}}$ that characterizes a class of frames $\mathfrak{C}$ by the property that $\vDash_{\mathfrak{C}} A$ if and only if $\mathsf{K} \wedge \mathsf{AX}_{\mathfrak{C}} \vDash A$.

Frame Condition	Description	Modal Axiom
Normal		$\mathsf{K} \coloneqq \Box(A \to B) \to (\Box A \to \Box B)$
Reflexive	wRw for all $w \in W$	$\mathsf{D} \coloneqq \Box A \to \Diamond A$
Serial	vRw for all $v \in W$ there exists $w \in W$	$\mathsf{T} \coloneqq \Box A \to A$
Transitive	uRv and vRw implies uRw for all $u, v, w \in W$	$\mathsf{4} \coloneqq \Box A \to \Box\Box A$
Euclidean	uRv and uRw implies vRw for all $u, v, w \in W$	$\mathsf{5} \coloneqq \Diamond A \to \Box\Diamond A$
Symmetry	wRv implies vRw for all $w, v \in W$	$\mathsf{B} \coloneqq A \to \Box\Diamond A$

For example, *Epistemic (modal) logic* is a class of logic that is used to reason about knowledge. In this logic, we consider sets of agents, each with their own knowledge, and we use the modal operator K_i to denote that the agent i knows a formula. The models of this logic are more complex, since each modal operator defines an accessibility relation R_i between worlds. Moreover, modalities in epistemic logic satisfying the axiom $\mathsf{S5} = \mathsf{T} \wedge \mathsf{4} \wedge \mathsf{5}$, which is the conjunction of the axioms for serial, transitive and euclidean.

Another interesting application of modal logic is *Dynamic logic*, where modalities are used to express actions and their effects. In **propositional dynamic logic**, each program α defines modalities $[\alpha]$ and $\langle\alpha\rangle$, and the accessibility relation R_α. The worlds of the model are interepreted as the states of the system, and the accessibility relation R_α connects the world v with w if the program α transforms the state v into the state w.

The formula $[\alpha]\, A$ can be interpreted as 'the formula A is true in any state which can be reached after executing α', while $\langle\alpha\rangle\, A$ is interpreted as 'the formula A is true in some state which can be reached after executing α'. Dynamic logics are at the base of the theory of *program verification*, where the goal is to check whether a program satisfies a given specification. Notable examples are the **modal mu-calculus** and the **Hennessy–Milner logic**.

5 First-Order Logic (W.I.P.)

6 Formal Reasoning in FoL Logic (W.I.P.)