P. Madhusudan, Mahesh Viswanathan

Logic in Computer Science

Rough course notes

September 14, 2021
# Contents

1 **Propositional Logic** ........................................... 1  
1.1 Syntax ......................................................... 3  
1.2 Semantics ..................................................... 6  
1.3 Satisfiability and Validity ................................. 9  
1.4 Compactness Theorem ................................. 16  
  1.4.1 Compactness using König’s Lemma ............... 16  
  1.4.2 Compactness using Henkin Models ............... 19  
  1.4.3 An Application of Compactness: Coloring Infinite Planar Graphs ........................................... 20  

2 **Proof Systems** ............................................. 23  
2.1 A Frege-style Proof System ................................. 24  
  2.1.1 Completeness Theorem .......................... 26  
2.2 Resolution .................................................. 27  
  2.2.1 Proving Tautologies with Resolution ........... 30  
  2.2.2 Completeness of Resolution ....................... 32  
2.3 Craig’s Interpolation Theorem and Proof Complexity ...... 34  
  2.3.1 Craig’s Interpolation Theorem ................. 34  
  2.3.2 Size of Interpolants ............................ 35  
  2.3.3 Interpolants from Refutations ................. 38  
  2.3.4 Lower bounds on Resolution Refutations ........ 44  

3 **First Order Logic** ........................................ 49  
3.1 Syntax ......................................................... 49  
3.2 Semantics ..................................................... 51  

A **Computability and Complexity Theory** ............. 55  
A.1 Turing Machines ......................................... 56  
A.2 Church-Turing Thesis .................................... 59  
A.3 Recursive and Recursively Enumerable Languages .......... 61  
A.4 Reductions .................................................. 63
A.5  Complexity Classes ................................................. 66
A.6  Relationship between Complexity Classes ......................... 69
A.7  P and NP ................................................................. 71
      A.7.1  Alternate characterization of NP  ......................... 72
      A.7.2  Reductions, Hardness and Completeness ................. 73
Chapter 1
Propositional Logic

Modern logic is a formal, symbolic system that tries to capture the principles of correct reasoning and truth. To describe any formal language precisely, we need three pieces of information — the alphabet describes the symbols used to write down the sentences in the language; the syntax describes the rules that must be followed for describing “grammatically correct” sentences in the language; and finally, the semantics gives “meaning” to the sentences in our formal language. You may have encountered other contexts where formal languages were introduced in such a manner. Here are some illustrative examples.

Example 1.1 Binomial coefficients are written using natural numbers and parentheses. However, not every way to put together parenthesis and natural numbers is a binomial coefficient. For example, \((1, 1)\), or \((\overline{7})\) are examples of things that are no binomial coefficients. Correctly formed binomial coefficients are of the form \:\binom{i+j}{i},\ where \(i\) and \(j\) are natural numbers. We could define the meaning of \:\binom{i+j}{i}\ to be the natural number \:\binom{i+j}{i}\. On the other hand, we could define the meaning of \:\binom{i+j}{i}\ to be the number of ways of choosing \(i\) elements from a set of \(i+j\) elements. Though both these ways of interpreting binomial coefficients are the same, they have a very different presentation. In general, one could define semantics in different ways, or even very different semantics to the same syntactic objects.

Example 1.2 Precise definitions of programming languages often involve characterizing its syntax and semantics. Turtle is an extremely simple programming language for drawing pictures. Programs in this language are written using \(\text{F, +, and } \overline{-}\). Any sequence formed by such symbols is a syntactically correct program in this language. We will interpret such a sequence of symbols as instructions to draw a picture — \(\text{F}\) is an instruction to draw a line by moving forward 1 unit; \(+\) is an instruction to turn the heading direction 60° to the left; \(\overline{-}\) is an instruction to turn the heading direction 60° to the right. Figure 1.1 shows example programs and the pictures they draw based on this interpretation.

Even though Turtle is a very simple programming language, some very interesting curves can be approximated. Consider the following iterative procedure that produces
a sequence of programs. Start with the program \( F \). In each iteration, if \( P \) is a program at the start of the iteration, then construct the program \( P' \) obtained by replacing each \( F \) in \( P \) by \( F + F - F + F \). So at the beginning we have program \( F \), in the next iteration the program is \( F + F - F + F \), and in the iteration after that it will be \( F + F - F + F + F - F + F - F + F - F + F + F - F + F \), and so on. The programs in this sequence draw pictures that in the limit approach the Koch curve.

Example 1.3 Regular expressions define special collections of strings called regular languages. Regular expressions over an alphabet \( \Sigma \) are built up using \( \Sigma \), parentheses, \( \emptyset \), \( \varepsilon \), \( * \), and \( + \). Inductively, they are defined as the smallest set that satisfy the following rules.

- \( \emptyset \) and \( \varepsilon \) are regular expressions.
- For any \( a \in \Sigma \), \( a \) is a regular expression.
- If \( r_1, r_2 \) are regular expressions then so are \( (r_1 \cdot r_2) \), \( (r_1 + r_2) \), and \( (r_1^*) \).

Each regular expression \( r \), semantically defines a subset of \( \Sigma^* \) that we will denote by \([r]\). The semantics of regular expressions is defined inductively as follows.

- \([\emptyset] = \emptyset \), and \([\varepsilon] = \{\varepsilon\}\).
- For \( a \in \Sigma \), \([a] = \{a\}\).
- Inductively, \([r_1 + r_2]\) = \([r_1]\) \( \cup \) \([r_2]\), \([r_1 \cdot r_2]\) = \([r_1]\) \( \cdot \) \([r_2]\) and \([r_1^*]\) = \([r_1]\)^*, where \( \cdot \) (on the right hand side) denotes the concatenation of two languages, and \( ^* \) denotes the Kleene closure of a language.

We will now define one of the simplest logics encountered in an introductory discrete mathematics class called propositional or sentential logic. Propositional logic is a symbolic language to reason about propositions. Propositions are declarative sentences that are either true or false. Examples of such include “Springfield is the capital of Illinois”, “\( 1+1 = 2 \)”, “\( 2+2 = 0 \)”. Notice that propositions don’t need to be true facts and their truth may depend on the context. For example, “\( 2+2 = 0 \)” is not true under the standard interpretation of \( + \) as integer addition, but is true if \( + \) denotes addition modulo \( 4 \). Non-examples of propositions include questions (like “What is it?”), commands (like “Read this!”), and things like “Location of robot”.

---

1 For a finite set \( \Sigma \), \( \Sigma^* \) denotes the collection of (finite) sequences/strings/words over \( \Sigma \). For \( n \in \mathbb{N} \), we use \( \Sigma^n \) to denote the set of sequences/strings/words over \( \Sigma \) of length exactly \( n \).
The logic itself will be symbolic and abstract away from English sentences like the ones above. We will introduce a precise definition of this logic, much in the same way as Example 1.3, defining the syntax and semantics inductively.

1.1 Syntax

We will assume a (countably infinite) set of propositions $\text{Prop} = \{ p_i \mid i \in \mathbb{N} \}$. The formulas of propositional logic will be strings over the alphabet $\text{Prop} \cup \{ (, ) , \neg , \lor \}$. Here $\neg$ is negation, and $\lor$ is disjunction.

**Definition 1.4** The set of well formed formulas (wff) in propositional logic (over the set $\text{Prop}$) is the smallest set satisfying the following properties.

1. Any proposition $p_i \in \text{Prop}$ is a wff.
2. If $\varphi$ is a wff then $(\neg \varphi)$ is a wff.
3. If $\varphi$ and $\psi$ are wffs then $(\varphi \lor \psi)$ is a wff.

Examples of wffs include $p_1 , (\neg p_1) , (p_1 \lor p_2) , ((\neg (p_1 \lor p_3)) \lor (p_1 \lor p_4))$. On the other hand the following strings are not wffs: $(p_1 \neg) , (\lor p_1) , (p_1 \lor)$.

Inductive definitions of the kind in Example 1.3 or Definition 1.4 are quite common when defining the syntax of formulas in a logic or of programming languages. Therefore, in computer science, one often uses a “grammar-like” presentation for the syntax. For example, wffs $\varphi$ in propositional logic are given by the following BNF grammar.

$$\varphi ::= p \mid (\neg \varphi) \mid (\varphi \lor \varphi)$$

(1.1)

where $p$ is an element of $\text{Prop}$. Reading such grammars takes some getting used to. For example, the rule $(\varphi \lor \varphi)$ doesn’t mean that disjunctions can only be used when the two arguments are the same. Instead it says that if we take two elements that belong to the syntactic entity $\varphi$ (i.e., wffs), put $\lor$ between them with surrounding parenthesis, then we get another element belonging to the same syntactic entity $\varphi$. We will sometimes use such a grammar representation to describe syntax in a succinct manner.

**Inductive Definitions**

What is the set identified by inductive definitions like Definition 1.4 and (1.1) ? Is there a unique single minimal set that satisfies the conditions in Definition 1.4? After all sets can be incomparable with respect to the $\subseteq$ relation. And what does the grammar given in (1.1) mean? Finally, do Definition 1.4 and (1.1) identify the same set?

Let us begin by first defining the set described by (1.1). Equation (1.1) defines the set $S = \bigcup_{i \in \mathbb{N}} S_i$, where the sets $S_i$ (for $i \in \mathbb{N}$) are given as follows.

$$S_0 = \text{Prop}$$

$$S_{i+1} = S_i \cup \{ (\neg \varphi) \mid \varphi \in S_i \} \cup \{ (\psi \lor \varphi) \mid \psi , \varphi \in S_i \} \quad \text{for } i \geq 0$$
Note that \( S = \bigcup_{i \in \mathbb{N}} S_i \) is an infinite union. You can think of \( S \) as the limit of the increasing sequence of sets \( S_0 \cup S_1 \cup \cdots \cup S_n \). Another way to think of \( S \) is as the set of all \( \varphi \) that belong to some \( S_i \). That is,
\[
S = \{ \varphi \mid \varphi \in S_i, \text{ for some } i \}.
\]

Another way to interpret the sets \( S \) and \( S_i \) is as follows. \( S_i \) denotes the set of formulas that can be derived from the grammar rules within \( i \) steps, and \( S \) is the set of formulas that can be derived from the grammar rules in some finite number of steps.

Given the above meaning, it’s natural to prove properties about the set of expressions \( S \) using induction on \( i \). More precisely, if we want to show a property \( P \) is true about \( S \), then we:

- Establish \( P \) to be true for every expression in \( S_0 \).
- For every \( i \geq 0 \), we assume that \( P \) holds for every expression in \( S_i \) and prove it holds for all expressions in \( S_{i+1} \).

The above shows that \( P \) holds on \( S_i \), for every \( i \in \mathbb{N} \), and hence \( P \) holds for every formula in \( S \).

Let us now consider Definition 1.4 and argue that it is well defined. Before showing that there is a unique smallest set satisfying conditions (1), (2), and (3) of Definition 1.4, let us argue that there is at least one set satisfying (1), (2), and (3). Take the set of all possible strings built over the alphabet \( \text{Prop} \cup \{ (, ) , \neg, \lor \} \). This set clearly satisfies all the conditions. But why should there be a smallest set? Observe that if two sets \( A \) and \( B \) satisfy the conditions in Definition 1.4, then so does \( A \cap B \). More generally, if \( \{ A_i \}_{i \in I} \) is a (possibly infinite) collection of sets satisfying (1), (2), and (3), then so does the set \( \bigcap_{i \in I} A_i \). Thus, the intersection of all sets which satisfy the conditions, is the unique, smallest (with respect to set inclusion) set identified by Definition 1.4. Hence, it is well defined.

Let \( S \) denote the set identified by grammar in (1.1) and \( T \) the set defined in Definition 1.4. We will argue that these two sets are the same. First, let us show that \( S \) satisfies conditions (1), (2), and (3) of Definition 1.4. Observe that, by definition, for every \( i \), \( S_i \subseteq S_{i+1} \). Hence, the sets increase with index, i.e., for every \( i < j \), \( S_i \subseteq S_j \); this can be formally established by induction, and we leave this as an exercise. Since \( \text{Prop} \subseteq S_0 \), this means \( \text{Prop} \subseteq S_i \) for every \( i \), and therefore \( S \) satisfies condition (1). Next, consider any \( \varphi, \psi \in S \). By definition of \( S \), this means there is are \( i, j \in \mathbb{N} \) such that \( \varphi \in S_i \) and \( \psi \in S_j \). Taking \( k = \max(i,j) \), we can conclude that \( \{ \varphi, \psi \} \subseteq S_k \). Thus, by definition, \( \neg \varphi \) and \( \varphi \lor \psi \) both belong to \( S_{k+1} \), and hence are in \( S \). Therefore, \( S \) satisfies conditions (2) and (3) of Definition 1.4. Finally, since \( S \) satisfies all conditions of Definition 1.4 and \( T \) is the smallest set with these properties, we can conclude that \( T \subseteq S \).

To prove the other inclusion (that \( S \subseteq T \)), we will prove that for every \( i \), \( S_i \subseteq T \) by induction. In the base case observe that \( S_0 = \text{Prop} \subseteq T \), since \( T \) satisfies condition (1). Assume as induction hypothesis, that for all \( j \leq i \), \( S_j \subseteq T \). In the induction step, we need to establish the claim that \( S_{i+1} \subseteq T \). Let \( \varphi \in S_{i+1} \) be an arbitrary element. By the definition of \( S_{i+1} \), there are 3 cases to consider. If \( \varphi \in S_i \) then by induction hypothesis \( \varphi \in T \). If \( \varphi = (\neg \psi) \) for some \( \psi \in S_i \), then by induction hypothesis \( \psi \in T \),
and since $T$ satisfies condition (2), $\varphi = (\neg \psi) \in T$. Finally, if $\varphi = (\psi_1 \lor \psi_2)$ for some $\psi_1, \psi_2 \in S$, then by induction hypothesis, $\psi_1, \psi_2 \in T$ and since $T$ satisfies condition (3), $\varphi = (\psi_1 \lor \psi_2) \in T$. This completes the proof that $S \subseteq T$, and therefore, $S = T$.

The above argument, establishing that the smallest set satisfying some closure conditions is the same as the set of objects derived from grammar rules, can be generalized to any grammar (not just context-free grammars). In general, if one defines a set as the smallest set $S$ that satisfies conditions of the form “if these elements belong to $S$ than these other elements must belong to $S'$”, then the smallest set is well-defined. But if you also have conditions saying “if these elements belong to $S$, these elements should not belong to $S'$,” then the “smallest” set may not be a well-defined.

Inductive or recursive definitions are ubiquitous in computer science, and reasoning about such definitions is naturally done using some form of induction. In fact, the prevalence of induction in computer science is because of the ubiquity of recursive definitions. Here are some other examples of recursive definitions.

- Consider the operational semantics of a program. The set of states/configurations that the program can reach is best defined recursively. It is the smallest set $R$ of states such that (a) $R$ contains the initial states of the program, and (b) if a state $s$ is in $R$ and the program can transition in one step from $s$ to $s'$, then $s' \in R$. Because of the recursive nature of this definition, reasoning about programs often involves induction. For example, to show that a program does not throw an exception, one needs to prove that any reachable state is one that does not throw an exception. This is typically established using induction.
- Consider lists in programs. The cons operator constructs a new list by adding an element to the head of another list. Lists over the elements $E$ can then be defined as the smallest set $L$ such that (a) $L$ contains nil, the empty list, and (b) if $\ell \in L$ and $e \in E$, then $\text{cons}(e, \ell)$ is in $L$ as well.
- The set of natural numbers can also be defined recursively. Let us denote by succ, the successor function $\mathbb{N}$. Then the set of natural numbers is the smallest set $\mathbb{N}$ such that (a) $\mathbb{N}$ contains 0, and (b) for every $n \in \mathbb{N}$, $\text{succ}(n)$ belongs to $\mathbb{N}$.

Other logical operators and Operator Precedence

Conjunction and implication are logical operators that arise quite often when expressing properties. These operators can be defined in terms of $\neg$ and $\lor$. Let $\varphi$ and $\psi$ be wffs, $(\varphi \land \psi)$ (read “$\varphi$ and $\psi$”) denotes the formula $(\neg((\neg\varphi) \lor (\neg\psi)))$. And $(\varphi \rightarrow \psi)$ (read “$\varphi$ implies $\psi$”) denotes the formula $((\neg\varphi) \lor \psi)$. Another useful wff is $\top$ (read “true”); it denotes the formula $(p \lor (\neg p))$, where $p$ is (any) proposition. Finally the wff $\bot$ (read “false”) is the formula $(\neg \top)$.

\[\begin{align*}
\top & = (p \lor (\neg p)) \\
\bot & = (\neg \top)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]

\[\begin{align*}
\varphi \land \psi & = (\neg((\neg\varphi) \lor (\neg\psi))) \\
\varphi \lor \psi & = (\neg((\neg\varphi) \land (\neg\psi))) \\
\neg \varphi & = ((\neg\varphi) \lor (\neg (\neg\varphi))) \\
\varphi \rightarrow \psi & = ((\neg\varphi) \lor \psi)
\end{align*}\]
Writing formulas strictly according to the syntax presented is cumbersome because of many parentheses and subscripts. Therefore, we will make the following notational simplifications.

- The outermost parentheses will be dropped. Thus we will write \( p_3 \lor (p_2 \lor p_1) \) instead of \( (p_3 \lor (p_2 \lor p_1)) \)
- We will sometimes omit subscripts of propositions. Thus we will write \( p \) instead of \( p_1 \), or \( q \) instead of \( p_2 \), \( r \) instead of \( p_3 \), or \( s \) instead of \( p_4 \), and so on.
- The following precedence of operators will be assumed: \( \neg, \land, \lor, \rightarrow \). Thus \( \neg p \land q \rightarrow r \) will mean \( (((\neg p) \land q) \rightarrow r) \).

Definition 1.4 for wffs in propositional logic has the nice property that the structure of a formula can be interpreted in a unique way. There is no ambiguity in its interpretation. For example, if \( \neg p_1 \lor p_2 \) were a wff, then it is unclear whether we mean the formula \( \varphi = ((\neg p_1) \lor p_2) \) or \( \psi = (\neg(p_1 \lor p_2)) \) — in \( \varphi \lor \) is the topmost operator, while in \( \psi \neg \) is the topmost operator. Our syntax does not have such issues. This will be exploited often in inductive definitions and in algorithms. This observation can be proved by structural induction, but we skip its proof.

**Theorem 1.5 (Unique Readability)**

Any wff can be uniquely read, i.e., it has a unique topmost logical operator and well defined immediate sub-formulas.

### 1.2 Semantics

We will now provide a meaning or semantics to the formulas. Our definition will follow the inductive definition of the syntax, just like in Example 1.3. The semantics of formulas in a logic, are typically defined with respect to a model, which identifies a “world” in which certain facts are true. In the case of propositional logic, this world or model is a truth valuation or assignment that assigns a truth value (true/false) to every proposition. The truth value truth will be denoted by \( T \), and the truth value falsity will be denoted by \( F \).

**Definition 1.6** A (truth) valuation or assignment is a function \( v \) that assigns truth values to each of the propositions, i.e., \( v : \text{Prop} \rightarrow \{T, F\} \).

The value of a proposition \( p \) under valuation \( v \) is given by \( v(p) \).

We will define the semantics through a satisfaction relation, which is a binary relation \( \models \) between valuations and formulas. The statement \( v \models \varphi \) should be read as “\( v \) satisfies \( \varphi \)” or “\( \varphi \) is true in \( v \)” or “\( v \) is a model of \( \varphi \)”. It is defined inductively following the syntax of formulas. In the definition below, we say \( v \not\models \varphi \) when \( v \models \varphi \) does not hold.

**Definition 1.7** For a valuation \( v \) and wff \( \varphi \), the satisfaction relation, \( v \models \varphi \), is defined inductively based on the structure of \( \varphi \) as follows.

---

3 For the formulas here, we are not using the precedence rules given before.
1.2 Semantics

- $v \models p$ if and only if $v(p) = T$.
- $v \models (\neg \varphi)$ if and only if $v \not\models \varphi$.
- $v \models (\varphi \lor \psi)$ if either $v \models \varphi$ or $v \models \psi$.

**Example 1.8** Let us look at a couple of examples to see how the inductive definition of the satisfaction relation can be applied. Consider the formula $\varphi = \neg(\neg p \lor \neg q) \lor (\neg p \lor \neg q)$. Recall with respect to the notational simplifications we identified, $\varphi$ is the formula $((\neg((\neg p) \lor (\neg q))) \lor ((\neg p) \lor (\neg q)))$. Consider the valuation $v_1$ that sets all propositions to $T$. Now $v_1 \models \varphi$ can be seen from the following observations.

- $v_1 \models p$ because $v_1(p) = T$
- $v_1 \not\models \neg p$ semantics of $\neg$
- $v_1 \models q$ because $v_1(q) = T$
- $v_1 \not\models \neg q$ semantics of $\neg$
- $v_1 \not\models \neg p \lor \neg q$ semantics of $\lor$
- $v_1 \models \neg(\neg p \lor \neg q)$ semantics of $\neg$
- $v_1 \models \neg(\neg p \lor \neg q) \lor (\neg p \lor \neg q)$ semantics of $\lor$

Consider $v_2$ that assigns all propositions to $F$. Once again $v_2 \models \varphi$. The reasoning behind this observation is as follows.

- $v_2 \not\models p$ because $v_2(p) = F$
- $v_2 \models \neg p$ semantics of $\neg$
- $v_2 \not\models \neg p \lor \neg q$ semantics of $\lor$
- $v_2 \models \neg(\neg p \lor \neg q) \lor (\neg p \lor \neg q)$ semantics of $\lor$

The semantics in Definition 1.7 defines a satisfaction relation between valuations and formulas. However, one could defined the semantics of propositional logic differently, by considering the formula as a “program” or “circuit” that computes a truth value based on the assignment. This approach is captured by the following definition of the value of a wff under a valuation.

**Definition 1.9** The value of a wff $\varphi$ under valuation $v$, denoted by $v[\varphi]$, is inductively defined as follows.

$$
\begin{align*}
    v[p] & = v(p) \\
    v[\neg \varphi] & = \begin{cases} 
        F & \text{if } v[\varphi] = T \\
        T & \text{if } v[\varphi] = F
    \end{cases} \\
    v[\varphi \lor \psi] & = \begin{cases} 
        F & \text{if } v[\varphi] = T \text{ or } v[\psi] = T \\
        T & \text{otherwise}
    \end{cases}
\end{align*}
$$

**Example 1.10** Let us consider $\varphi = \neg(\neg p \lor \neg q) \lor (\neg p \lor \neg q)$ and $v_1$ which assigns all propositions to $T$. $v_1[\varphi]$ can be computed as follows.
\[ \begin{align*}
\mathcal{V} &_{p} = T \\
\mathcal{V} \neg p & = F \\
\mathcal{V} q & = T \\
\mathcal{V} \neg q & = F \\
\mathcal{V} p \lor \neg q & = F \\
\mathcal{V} (p \lor \neg q) \lor (p \land \neg q) & = T
\end{align*} \]

because \( \mathcal{V}(p) = T \)  
semantics of \( \neg \)  
semantics of \( \lor \)  
semantics of \( \land \)  
semantics of \( \lor \)  
semantics of \( \land \)

Definitions 1.7 and 1.9 are both equivalent in some sense. This is captured by the following theorem.

**Theorem 1.11** For any truth valuation \( \mathcal{V} \) and wff \( \phi \), \( \mathcal{V} \models \phi \) if and only if \( \mathcal{V}[\phi] = T \)

The proof of Theorem 1.11 is by structural induction on the formula \( \phi \). It is left as an exercise for the reader.

It is convenient to associate with every wff the set of truth valuations under which the formula holds.

**Definition 1.12** The *models* of wff \( \phi \) is the set of valuations that *satisfy* \( \phi \). More precisely,

\[ [\phi] = \{ \mathcal{V} | \mathcal{V} \models \phi \} \]

Observe that as per the definition, \([\bot] = \emptyset \).

The *relevance lemma* says that whether \( \phi \) holds under a valuation depends only on how the valuation maps the propositions that syntactically occur in the formula. This is intuitively obvious; surely, whether \( (p \land q) \lor r \) holds is independent of whether the proposition \( s \) is mapped to true/false. For a wff \( \phi \), the set of propositions appearing in \( \phi \), denoted \( \text{occ}(\phi) \), is inductively defined as follows.

- \( \text{occ}(p) = \{ p \} \)
- \( \text{occ}(\neg \phi) = \text{occ}(\phi) \)
- \( \text{occ}(\phi \land \psi) = \text{occ}(\phi) \cup \text{occ}(\psi) \)

The relevance lemma is then as follows.

**Lemma 1.13 (Relevance Lemma)**

Let \( \mathcal{V}_1 \) and \( \mathcal{V}_2 \) be truth valuations such that for all \( p \in \text{occ}(\phi) \), we have \( \mathcal{V}_1(p) = \mathcal{V}_2(p) \), i.e., \( \mathcal{V}_1 \) and \( \mathcal{V}_2 \) agree on the truth values assigned to all propositions in \( \text{occ}(\phi) \). Then \( \mathcal{V}_1 \models \phi \) if and only if \( \mathcal{V}_2 \models \phi \).

**Proof** By structural induction on \( \phi \).

**Base Case** \( \phi = p \) Observe that, \( \mathcal{V}_1 \models \phi \) iff \( \mathcal{V}_1(p) = T = \mathcal{V}_2(p) \) iff \( \mathcal{V}_2 \models \phi \).

**Induction Step** \( \phi = (\neg \psi) \) Since \( \text{occ}(\psi) \subseteq \text{occ}(\phi) \), we have by induction hypothesis, \( \mathcal{V}_1 \models \psi \) iff \( \mathcal{V}_2 \models \psi \). Therefore, by the semantics of \( \neg \), \( \mathcal{V}_1 \models \phi \) iff \( \mathcal{V}_2 \models \phi \).

**Induction Step** \( \phi = (\psi_1 \lor \psi_2) \) Since \( \text{prop}(\psi_i) \subseteq \text{prop}(\phi) \) (for \( i \in \{1, 2\} \)), we have by induction hypothesis, \( \mathcal{V}_1 \models \psi_i \) iff \( \mathcal{V}_2 \models \psi_i \). Therefore, by the semantics of \( \lor \), \( \mathcal{V}_1 \models \phi \) iff \( \mathcal{V}_2 \models \phi \). \( \square \)
Lemma 1.13 implies that, to determine if a formula holds in a valuation, we only need to consider the assignment to the finitely many propositions occurring in the formula. Thus, instead of thinking of valuations as assigning truth values to all (infinitely many) propositions, we can think of them as functions with a finite domain.

### 1.3 Satisfiability and Validity

Two formulas that are syntactically different, could however, be “semantically equivalent”. But what do we mean by semantic equivalence? Intuitively, this is when the truth value of each formula in every valuation is the same.

**Definition 1.14 (Logical Equivalence)**

A wff \( \varphi \) is said to be **logically equivalent to** \( \psi \) iff any of the following equivalent conditions hold.

- for every valuation \( v \), \( v \models \varphi \text{ iff } v \models \psi \),
- for every valuation \( v \), \( v[\varphi] = v[\psi] \),
- \( \llbracket \varphi \rrbracket = \llbracket \psi \rrbracket \).

We denote this by \( \varphi \equiv \psi \).

Let us consider an example to see how we may reason about two formulas being logically equivalent.

**Example 1.15** Consider the wffs \( \varphi_1 = p \land (q \lor r) \) and \( \varphi_2 = (p \land q) \lor (p \land r) \), where \( p, q \) and \( r \) are propositions. Though \( \varphi_1 \) and \( \varphi_2 \) are syntactically different, they are semantically equivalent. To prove that \( \varphi_1 \equiv \varphi_2 \), we need to show that they two formulas evaluate to the same truth value under every valuation. One convenient way to organize such a proof is as truth table, where different cases in the case-by-case analysis correspond to different rows. Each row of the truth table corresponds to a (infinite) collection of valuations based on the value assigned to propositions \( p, q \) and \( r \); the columns correspond to the value of different (sub)-formulas under each valuation in this collection. For example, a truth table reasoning for \( \varphi_1 \) and \( \varphi_2 \) will look as follows.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( r )</th>
<th>( q \lor r )</th>
<th>( \varphi_1 )</th>
<th>( p \land q )</th>
<th>( p \land r )</th>
<th>( \varphi_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Notice that since the columns corresponding to $\varphi_1$ and $\varphi_2$ are identical in every row, and every valuation corresponds to some row in the table, it follows that $\varphi_1$ and $\varphi_2$ are logically equivalent.

Let us now consider $\varphi'_1 = \psi_1 \land (\psi_2 \lor \psi_3)$ and $\varphi'_2 = (\psi_1 \land \psi_2) \lor (\psi_1 \land \psi_3)$, where $\psi_1$, $\psi_2$ and $\psi_3$ are arbitrary wffs. Once again $\varphi'_1$ and $\varphi'_2$ are logically equivalent, no matter what $\psi_1$, $\psi_2$ and $\psi_3$ are. The reasoning is essentially the same as above. The rows of the truth table now classify valuations based on the value of formulas $\psi_1$, $\psi_2$ and $\psi_3$ under them.

<table>
<thead>
<tr>
<th>$\psi_1$</th>
<th>$\psi_2$</th>
<th>$\psi_3$</th>
<th>$\psi_2 \lor \psi_3$</th>
<th>$\varphi_1$</th>
<th>$\psi_1 \land \psi_2$</th>
<th>$\psi_1 \land \psi_3$</th>
<th>$\varphi_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Truth table based reasoning, as carried out in Example 1.15, is a very convenient way to organize proofs of propositional logic. We will often use it. Example 1.15 highlights another important observation. Let $\varphi$ and $\psi$ be logically equivalent formulas. Let $\varphi'$ and $\psi'$ be formulas obtained by substituting propositions occurring in $\varphi$ and $\psi$ by arbitrary formulas. Then $\varphi' \equiv \psi'$.

**Definition 1.16 (Logical Consequence)**

Let $\Gamma$ be a (possibly infinite) set of formulas and let $\varphi$ be a wff. We say that $\varphi$ is a logical consequence of $\Gamma$ (denoted $\Gamma \models \varphi$) iff for every valuation $v$, if for every $\psi \in \Gamma$, $v \models \psi$ then $v \models \varphi$. In other words, any model that satisfies every formula in $\Gamma$ also satisfies $\varphi$.

We could equivalently have defined it as $\Gamma \models \varphi$ iff $\bigwedge_{\psi \in \Gamma} [\models \psi] \subseteq [\models \varphi]$.

**Example 1.17** Consider the set $\Gamma = \{ \psi_1 \rightarrow \psi_2, \psi_2 \rightarrow \psi_1, \psi_1 \vee \psi_2 \}$, where $\psi_1$ and $\psi_2$ are arbitrary formulas. We will show that $\Gamma \models \psi_1$. Once again, we will use a truth table to classify valuations into row based on the value that $\psi_1$ and $\psi_2$ evaluate to.

Such a truth table looks as follows.

<table>
<thead>
<tr>
<th>$\psi_1$</th>
<th>$\psi_2$</th>
<th>$\psi_1 \rightarrow \psi_2$</th>
<th>$\psi_2 \rightarrow \psi_1$</th>
<th>$\psi_1 \vee \psi_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Notice that there is only one row where columns 3, 4, and 5 are all $T$; this corresponds the valuations where $\psi_1$ and $\psi_2$ evaluate to $T$, and under every such valuation, all formulas in $\Gamma$ are satisfied. In this row, since $\psi_1$ also evaluates to $T$, we have that $\Gamma \models \psi_1$. 
It is worth observing one special case of Definition 1.16 — when $\Gamma = \emptyset$. In this case, every valuation satisfies every formula in $\Gamma$ (vacuously, since there are none to satisfy). Therefore, if $\emptyset \models \varphi$, then every truth assignment satisfies $\varphi$. Such formulas are called tautologies, and they represent universal truths that hold in every model/world/assignment.

**Definition 1.18 (Tautologies)**

A wff $\varphi$ is a tautology or is valid if for every valuation $v$, $v \models \varphi$. In other words, $\emptyset \models \varphi$. We will denote this simply as $\models \varphi$.

*Example 1.19* We will show $\varphi = \psi_1 \rightarrow (\psi_2 \rightarrow \psi_1)$ is a tautology, no matter what formulas $\psi_1$ and $\psi_2$ are. The proof is once again organized as truth table, and we show that in all rows the formula $\varphi$ evaluates to $T$.

<table>
<thead>
<tr>
<th>$\psi_1$</th>
<th>$\psi_2$</th>
<th>$\psi_2 \rightarrow \psi_1$</th>
<th>$\psi_1 \rightarrow (\psi_2 \rightarrow \psi_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

The last important notion we would like to introduce is that of satisfiability.

**Definition 1.20 (Satisfiability)**

A formula $\varphi$ is satisfiable if there is some valuation $v$ such that $v \models \varphi$. In other words, $\models \varphi$. If a formula is not satisfiable, we say it is unsatisfiable.

*Example 1.21* $\varphi = (p \lor q) \land (\neg p \lor \neg q)$ is satisfiable because the valuation $v$ that maps $p$ to $T$ and $q$ to $F$ satisfies $\varphi$, i.e., $v \models \varphi$.

Based on Definitions 1.18 and 1.20, it is easy to see that there is a close connection between satisfiability and validity.

**Proposition 1.22** A wff $\varphi$ is valid if and only if $\neg \varphi$ is unsatisfiable.

**Proof** Let $v$ be any valuation. If $\varphi$ is valid, we know that $v \models \varphi$. Therefore, by the semantics of $\neg$, we have $v \models \neg \varphi$. Thus $\neg \varphi$ is unsatisfiable. Conversely, if $\neg \varphi$ is unsatisfiable, then $v \not\models \neg \varphi$. Again, by the semantics of $\neg$, $v \models \varphi$. Thus, $\varphi$ is valid. $\square$

We conclude this section by considering two fundamental computational problems — satisfiability and validity.

**Satisfiability** Given a formula $\varphi$, determine if $\varphi$ is satisfiable.

**Validity** Given a formula $\varphi$, determine if $\varphi$ is a tautology.

The satisfiability and validity problems have very simple algorithms to solve them. To check if $\varphi$, over propositions $\{p_1, \ldots, p_n\}$, is satisfiable (is a tautology), compute $v[\varphi]$ for every truth assignment $v$ to the propositions $\{p_1, \ldots, p_n\}$. The running time for this algorithm is $O(2^n)$. One of the most important open questions in computer science is whether this is the best algorithm for these problems. The following theorem by Cook and Levin, supports the belief that this exponential algorithm is unlikely to be improved in the worst case.
Theorem 1.23 (Cook-Levin)

The satisfiability problem for propositional logic is \( \text{NP-complete} \).

**Proof** Any proof showing a problem to be \( \text{NP-complete} \) has two parts. First is an argument that the problem belongs to \( \text{NP} \) and the second that it is hard.

**Membership in \( \text{NP} \).** Given a formula \( \varphi \), the \( \text{NP} \)-algorithm to check satisfiability is as follows — Guess a truth assignment \( v \), evaluate \( \varphi \) on the truth assignment, and accept if \( v[\varphi] = T \); otherwise reject. Guessing (nondeterministically) a truth assignment takes time which is linear in the number of propositions in \( \varphi \), which is linear in the size of \( \varphi \), and computing \( v[\varphi] \) also takes time that is linear in the size of \( \varphi \), where the evaluation algorithm computes the value in a “bottom-up” fashion. Thus, the total running time is polynomial.

**NP-hardness.** Consider \( A \in \text{NP} \). Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \sqcup, \triangleright) \) be a non-deterministic TM recognizing \( A \) in time \( n^t \), where \( Q \) is the set of control states, \( \Sigma \) is the input alphabet, \( \Gamma \) is the tape alphabet, \( \delta \) is the transition function, \( q_0 \) is the initial state, \( q_{\text{acc}} \) is the unique accepting state, \( q_{\text{rej}} \) is the unique rejecting state, \( \sqcup \) is the blank symbol, and \( \triangleright \) is the left end marker symbol that appears on the leftmost cell of each tape. Without loss of generality, we assume that \( q_{\text{acc}} \) and \( q_{\text{rej}} \) are the only halting states of \( M \). We will also assume that \( M \) has a read only input tape, and a single read/write work tape.

For an input \( x \), we will construct (in polynomial time) a formula \( f_M(x) \) such that \( M \) accepts \( x \) (i.e., \( x \in A \)) iff \( f_M(x) \) is satisfiable. \( f_M(x) \) will encode constraints on a computation of \( M \) on \( x \) such that a satisfying assignment to \( f_M(x) \) will describe “how \( M \) accepts \( x \)”. That is, \( f_M(x) \) will encode that

- \( M \) starts in the initial configuration with input \( x \),
- Each configuration follows from the previous one in accordance with the transition function of \( M \),
- The accepting state is reached in the last step.

Let us formalize this intuition by giving a precise construction. We begin by identifying the set of propositions we will use, and their informal interpretation.

**Propositional Variables.** The propositions of \( f_M(x) \) will be as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning if set to ( T )</th>
<th>Total Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>InpSymb@((b, p))</td>
<td>Input tape stores ( b ) at position ( p )</td>
<td>( O(</td>
</tr>
<tr>
<td>TapeSymb@((b, p, i))</td>
<td>Work tape stores ( b ) in cell ( p ) at time ( i )</td>
<td>( O(</td>
</tr>
<tr>
<td>InpHd@((h, i))</td>
<td>Input head in cell ( h ) at time ( i )</td>
<td>( O(</td>
</tr>
<tr>
<td>TapeHd@((h, i))</td>
<td>Work tape’s head in cell ( h ) at time ( i )</td>
<td>( O(</td>
</tr>
<tr>
<td>St@((q, i))</td>
<td>State is ( q ) at time ( i )</td>
<td>( O(</td>
</tr>
</tbody>
</table>

**Abbreviations.** In order to define \( f_M(x) \), the following abbreviations will be useful.

- \( \bigwedge_{k=1}^{m} X_k \) means \( X_1 \wedge X_2 \wedge \cdots \wedge X_m \)
1.3 Satisfiability and Validity

- \( \nabla(X_1, X_2, \ldots X_m) \) will denote a formula that is satisfiable iff exactly one of \( X_1, \ldots X_m \) is set to true. In other words,

\[
\nabla(X_1, X_2, \ldots X_m) = (X_1 \lor X_2 \lor \cdots \lor X_m) \land \bigwedge_{k \neq l} (\neg X_k \lor \neg X_l)
\]

**Overall Reduction.** The overall form of \( f_M(x) \) will be as follows.

\[
f_M(x) = \varphi_{\text{initial}} \land \varphi_{\text{consistent}} \land \varphi_{\text{transition}} \land \varphi_{\text{accept}}
\]

where

- \( \varphi_{\text{initial}} \) says that “configuration at time 0 is the initial configuration with input \( x \)”
- \( \varphi_{\text{consistent}} \) says that “at each time, truth values to variables encode a valid configuration”
- \( \varphi_{\text{transition}} \) says that “configuration at each time follows from the previous one by taking a transition”
- \( \varphi_{\text{accept}} \) says that “the last configuration is an accepting configuration”

We now outline what each of the above formulas is.

**Initial Conditions** Let \( x = a_1a_2\cdots a_n \)

\[
\varphi_{\text{initial}} = \text{St}@(q_0, 0) \land \text{InpSymb}@(\succ, 0, 0) \land \text{TapeSymb}@(\succ, 0, 0) \land \left( \bigwedge_{p=1}^{n} \text{InpSymb}@(a_p, p, 0) \land \text{TapeSymb}@(\perp, p, 0) \right)
\]

- “At time 0, state is \( q_0 \)”
- “Leftmost cells contain >”
- “At time 0, cells 1 through \( n \) hold \( x \)”
- “At time 0, all work tape cells are blank”
- “At time 0, all heads at the leftmost position”

**Consistency** Assume that the tape alphabet is \( \Gamma = \{b_1, b_2, \ldots b_t\} \) and the set of states is \( Q = \{q_0, q_1, \ldots q_m\} \).
Transition Consistency  Consider a non-halting state $q$ (i.e., $q \notin \{q_{\text{acc}}, q_{\text{rej}}\}$), input symbol $c_{\text{in}}$ and tape symbol $c_{\text{w}}$. Let the transition at state $q$, when reading these symbols be given by

$$
\delta(q, c_{\text{in}}, c_{\text{w}}) = \{(q^{(1)}, d_{\text{in}}^{(1)}, c_{\text{w}}^{(1)}, d_{\text{w}}^{(1)}), \ldots, (q^{(s)}, d_{\text{in}}^{(s)}, c_{\text{w}}^{(s)}, d_{\text{w}}^{(s)})\}.
$$

Here $(q^{(i)}, d_{\text{in}}^{(i)}, c_{\text{w}}^{(i)}, d_{\text{w}}^{(i)}) \in \delta(q, c_{\text{in}}, c_{\text{w}})$ means that if $M$ is in state $q$ and reads $c_{\text{in}}$ on the input tape and $c_{\text{w}}$ on the work tape, then one possible transition is to state $q^{(i)}$, moving the input head in direction $d_{\text{in}}^{(i)}$, writing $c_{\text{w}}^{(i)}$ on the work tape, and moving the work tape head in direction $d_{\text{w}}^{(i)}$. Direction $-1$ denotes moving the head left and $+1$ denotes moving the head right. We will first define a formula $\Delta_{q, c_{\text{in}}, c_{\text{w}}}^{l}$ that says that at time $i$ if the state is $q$ and the symbol read on the input tape is $c_{\text{in}}$ and on the work tape is $c_{\text{w}}$ at positions $p_{\text{in}}, p_{\text{w}}$, respectively, then at time $i + 1$ the state, symbols written and new head position is one of the tuples described by the $\delta$ function.

$$
\Delta_{q, c_{\text{in}}, c_{\text{w}}}^{l} = \left(\text{St}(q, i) \land \text{InpHd}(p_{\text{in}}, i) \land \text{InpSymb}(c_{\text{in}}, p_{\text{in}}, i) \land \text{TapeHd}(p_{\text{w}}, i) \land \text{TapeSymb}(c_{\text{w}}, p_{\text{w}}, i) \rightarrow \forall C_{\text{hi}}(i, p_{\text{in}}, p_{\text{w}}, \ldots C_{\text{hi}}(i, p_{\text{in}}, p_{\text{w}}))\right)
$$

where

$$
C_{\text{hi}}(i, p_{\text{in}}, p_{\text{w}}) = \text{St}(q^{(i)}, i + 1) \land \text{InpHd}(p_{\text{in}} + d_{\text{in}}^{(i)}, i + 1) \land \text{TapeSymb}(c_{\text{w}}^{(i)}, p_{\text{w}}, i + 1) \land \text{TapeHd}(p_{\text{w}} + d_{\text{w}}^{(i)}, i + 1)
$$

For a halting state $q$ (i.e., $q \in \{q_{\text{acc}}, q_{\text{rej}}\}$), we define $\Delta_{q, c_{\text{in}}, c_{\text{w}}}^{f}$ as saying that the state, symbols, and head positions don’t change. In other words,

$$
\Delta_{q, c_{\text{in}}, c_{\text{w}}}^{f} = \left(\text{St}(q, i) \land \text{InpHd}(p_{\text{in}}, i) \land \text{InpSymb}(c_{\text{in}}, p_{\text{in}}) \land \text{TapeHd}(p_{\text{w}}, i) \land \text{TapeSymb}(c_{\text{w}}, p_{\text{w}}, i) \rightarrow \text{St}(q, i + 1) \land \text{InpHd}(p_{\text{in}}, i + 1) \land \text{TapeSymb}(c_{\text{w}}, p_{\text{w}}, i + 1) \land \text{TapeHd}(p_{\text{w}}, i + 1)\right)
$$

when $q$ is a halting state.
Now Transition Consistency itself can be defined as follows.

$$
\varphi_{\text{transition}} = \bigwedge_{i=0}^{n'} \bigwedge_{p_n=0}^{n'} \bigwedge_{p_w=0}^{n'} \left\{ \right.

\land_{b,c} \neg \text{TapeHd}@(p_w, i) \rightarrow \\
\neg (\text{TapeSymb}@(b, p_w, i) \land \text{TapeSymb}@(c, p_w, i + 1))

\left. \right\}

\text{“If head is not in some position, then symbol does not change”}

\land_{q_{m'p}, \neg \text{q}_{m'p}, b, \neg b, c, \neg c} \land \text{q}_{m', p, p_w}

\text{“If head is in some position, then a transition is taken”}

\text{Acceptance}

$$

\varphi_{\text{accept}} = \text{St}@(a, n')$

We can argue that $M$ accepts $x$ if and only if $f_M(x)$ is satisfiable. Further $f_M(x)$ can be constructed in time that is polynomial in the size of $x$; the size of $M$ also plays a role but that is fixed.

The Cook-Levin Theorem (Theorem 1.23) is an important result in computer science. It was the first result establishing the intractibility of a problem. Moreover, it also implies the intractibility of the validity problem. This is because there is a formula $\varphi$ is valid if and only if $\neg \varphi$ is unsatisfiable.

**Proposition 1.24** A formula $\varphi$ is valid if and only if $\neg \varphi$ is unsatisfiable.

**Proof** The proposition can be established by the following sequence of observations. $\varphi$ is valid iff for every valuation $v$, $v[\varphi] = T$ (definition of validity) iff for every valuation $v$, $v[\neg \varphi] = F$ (from the semantics of $\neg$) iff $\neg \varphi$ is unsatisfiable (definition of unsatisfiability). □

Using Proposition 1.24 we establish the $\text{coNP}$-hardness of validity.

**Theorem 1.25** The validity problem for propositional logic is $\text{coNP}$-complete.

**Proof** Observe that there is a simple $\text{NP}$ algorithm to check that a formula $\varphi$ is not valid — Guess a valuation $v$, and check that $v[\varphi] = F$. Since checking non-validity has a $\text{NP}$ algorithm, it means that the validity problem has a $\text{coNP}$ algorithm, and is therefore, in $\text{coNP}$.

To prove the $\text{coNP}$-hardness of the validity problem, we make the following observations. First, for any two problems $A$ and $B$, if $A \leq_p B$ then $\overline{A} \leq_p \overline{B}$. Thus, from the $\text{NP}$-hardness of the satisfiability problem (Theorem 1.23), we can conclude that the problem of checking if a formula is unsatisfiable is $\text{coNP}$-hard. Since unsatisfiability is $\text{coNP}$-hard, it follows that validity is also $\text{coNP}$-hard based on the observation in Proposition 1.24. □
1.4 Compactness Theorem

The compactness theorem is an important property about propositional logic. In this section, we will look at a couple of different proofs of this theorem.

A (finite or infinite) set of formulas $\Gamma$ is satisfiable if there is a valuation $v$ such that for every $\varphi \in \Gamma$, $v \models \varphi$ (or $v[\varphi] = T$); we will denote this by $v \models \Gamma$. A set of formulas $\Gamma$ is finitely satisfiable if every finite subset $\Gamma_0$ of $\Gamma$ is satisfiable. These two notions, satisfiability and finite satisfiability, are equivalent — this is the content of the compactness theorem.

**Theorem 1.26 (Compactness)**

A set of formulas $\Gamma$ is satisfiable if and only if $\Gamma$ is finitely satisfiable.

Observe that if $\Gamma$ is satisfiable then the satisfying assignment (say $v$) also satisfies every subset of $\Gamma$ and therefore also every finite subset of $\Gamma$. Thus if $\Gamma$ is satisfiable then it is also finitely satisfiable. The challenge is, therefore, in proving the converse — that finite satisfiability implies satisfiability. If $\Gamma$ is a finite set, then clearly finite satisfiability implies satisfiability because $\Gamma$ itself is a finite subset of $\Gamma$. So the interesting case is when $\Gamma$ is infinite. We will provide a couple of very different proofs for this result.

Before moving on to present our proofs for Theorem 1.26, we highlight an important consequence of the theorem.

**Corollary 1.27** Let $\Gamma$ be a (possibly infinite) set of formulas and let $\varphi$ be a formula. If $\Gamma \models \varphi$ then there is a finite subset $\Delta \subseteq \Gamma$ such that $\Delta \models \varphi$.

**Proof** Let $\Gamma \models \varphi$. Then $\Gamma \cup \{\neg \varphi\}$ is unsatisfiable. By Theorem 1.26, there is a finite subset $\Delta' \subseteq \Gamma \cup \{\neg \varphi\}$ that is unsatisfiable. Taking $\Delta = \Delta' \setminus \{\neg \varphi\}$, we observe that $\Delta \subseteq \Gamma$ and $\Delta \cup \{\neg \varphi\}$ is unsatisfiable. Thus, $\Delta \models \varphi$. \qed

It is useful to note that the above argument works even when $\neg \varphi \notin \Delta'$.

1.4.1 Compactness using König’s Lemma

We present a simple and elegant proof of the compactness theorem that uses König’s Lemma. This proof approach works only for propositional logic, and does not extend to first order logic. Let us begin by recalling König’s lemma for binary trees.

A binary tree is said to have paths of arbitrary length if for each natural number $n$, there is a path in the tree whose length is $\geq n$. An infinite path in the binary tree is an infinite sequence of vertices of the tree such that successive vertices in the sequence are connected by an edge. Observe that if a binary tree has an infinite path then it also has paths of arbitrary length. This is because for every $n$, the prefix of the infinite path with $n + 1$ vertices, is a path in the binary tree of length $n$. König’s Lemma says that the converse of this is also true.
Lemma 1.28 (König)

A binary tree with paths of arbitrary length has an infinite path.

Proof Suppose \( u \) is a vertex that does not have paths of arbitrary length starting from it, then by definition, there must be a number \( m \) such that all paths starting from \( u \) are of length at most \( m \). Now, if a vertex \( u \) has the property that none of its children \( v \) have paths of arbitrary length starting from them, then \( u \) also cannot have paths of arbitrary length starting from it. The contrapositive of this statement is that if \( u \) is vertex with paths of arbitrary length starting from it, then at least one of its children \( v \) also has paths of arbitrary length.

Suppose a binary tree has paths of arbitrary length. Then the root is a vertex that has paths of arbitrary length starting from it. The infinite path is given by \( v_0, v_1, \ldots \) where \( v_0 \) is the root. \( v_{i+1} \) is the left child of \( v_i \), if the left child has paths of arbitrary length, and \( v_{i+1} \) is the right child of \( v_i \) otherwise.

Let us fix the set of propositions in our logic to be \( \text{Prop} = \{ p_i \mid i \in \mathbb{N} \} \). Truth assignments to \( \text{Prop} \) can be thought of as (infinite) paths in the complete, infinite binary tree — vertices at level \( i \) correspond to proposition \( p_i \) and if the path takes the left child at level \( i \), then it corresponds to the assignment setting \( p_i \) to 0; otherwise it sets \( p_i \) to 1. Finite paths in this tree, correspond to partial assignments. So a path of length \( i \) corresponds to an assignment that sets values to propositions \( \{ p_0, p_1, \ldots, p_{i-1} \} \).

For example, in Fig. 1.2, the bold path corresponds to the (partial) assignment \( \nu \) that sets \( \nu(p_0) = T, \nu(p_1) = F, \) and \( \nu(p_2) = T \). Recall that, whether a formula \( \phi \) holds in an assignment, depends only on the truth values assigned to the propositions that appear in \( \phi \) (Lemma 1.13). Thus, partial assignments can determine the truth of formulas that only mention the propositions that have been assigned. For example, the partial assignment indicated by the bold path in Fig. 1.2 can determine the truth of any formula that only mentions \( p_0, p_1, \) and \( p_2 \).
Proof (Of Theorem 1.26) Let $\Gamma$ be a set of formulas that is finitely satisfiable. Logical equivalence ($\equiv$) partitions $\Gamma$ into equivalence classes. Take $\Gamma'$ to be a subset of $\Gamma$ that contains exactly one representative from each equivalence class. That is, $\Gamma' \subseteq \Gamma$ such that

- For every $\varphi \neq \psi \in \Gamma'$, $\varphi \neq \psi$, and
- For every $\varphi \in \Gamma$, there is $\psi \in \Gamma'$ such that $\varphi \equiv \psi$.

Since $\Gamma' \subseteq \Gamma$, $\Gamma'$ is also finitely satisfiable.

Recall that $\operatorname{occ}(\varphi)$ is defined to be the set of propositions that appear in $\varphi$. For $i \geq 0$, define $\Gamma_i$ to be

$$\Gamma_i = \{ \varphi \in \Gamma' \mid \operatorname{occ}(\varphi) \subseteq \{ p_0, p_1, \ldots, p_{i-1} \} \}.$$ 

Observe that $\Gamma_i$ defines an non-decreasing sequence of sets, i.e., for every $i$, $\Gamma_i \subseteq \Gamma_{i+1}$. Also, $\Gamma = \bigcup_{i \geq 0} \Gamma_i$. The most important observation about $\Gamma_i$ is that it is a finite set — since $\Gamma'$ has only one formula from each equivalence class of $\equiv$, each formula in $\Gamma_i$ corresponds to a unique subset of assignments of $\{ p_0, \ldots, p_{i-1} \}$ to $\{ \text{F}, \text{T} \}$. Thus, we have $|\Gamma_i| \leq 2^i$. Since $\Gamma'$ is finitely satisfiable, $\Gamma_i$ is satisfiable for every $i$.

Define $T_\Gamma$ as the following set of (partial) truth assignments.

$$T_\Gamma = \bigcup_{i \geq 0} \{ v : \{ p_0, \ldots, p_{i-1} \} \to \{ \text{F}, \text{T} \} \mid v \models \Gamma_i \}.$$ 

Recall that we say $v \models \Gamma_i$ if $v$ satisfies all formulas in $\Gamma_i$. Consider an assignment $v \in T_\Gamma$ with domain $\{ p_0, p_1, \ldots, p_{i-1} \}$. By definition $v \models \Gamma_i$. For $j < i$, since $\Gamma_j \subseteq \Gamma_i$, $v \models \Gamma_j$. Further, $\Gamma_j$ only has propositions $\{ p_0, \ldots, p_{j-1} \}$, we also have $v' \models \Gamma_j$, where $v'$ is the restriction of $v$ to the domain $\{ p_0, \ldots, p_{j-1} \}$. So $v' \in T_\Gamma$.

Viewed as paths in the infinite binary tree (see Fig. 1.2.), $v$ is a path of length $i$, $v'$ its prefix of length $j$. What we observe is that $T_\Gamma$ is “closed” under prefix of paths. Thus, if we restrict our attention to the assignments in $T_\Gamma$ then they form a subtree of the infinite binary tree.

Let us consider $T_\Gamma$. In the previous paragraph we observed that it forms a subtree of the infinite binary tree. It has paths of arbitrary length; this is because every $\Gamma_i$ is satisfiable, and an (partial) assignment satisfying $\Gamma_i$ is a path of length $i$ in the tree. Since $T_\Gamma$ is a binary tree, by König’s lemma, it has an infinite path. The infinite path corresponds to a (full) truth assignment, say $v_*$. Further, since every prefix of $v_*$ is a (finite) path in $T_\Gamma$, it means that the prefix of length $i$ (viewed as a partial assignment) satisfies $\Gamma_i$. Therefore, for every $i$, $v_* \models \Gamma_i$, and hence $v_* \models \Gamma'$. Now, since every formula $\varphi \in \Gamma$ is logically equivalent to some formula $\psi \in \Gamma'$, it means $v_* \models \Gamma$. Thus, $\Gamma$ is satisfiable. \qed
1.4.2 Compactness using Henkin Models

The proof of the compactness theorem we present in the section, relies on constructing a truth assignment for a set of formulas $\Gamma$ through the process of saturation, where we add formulas to the set $\Gamma$ as long as it remains finitely satisfiable. This is an approach proposed by Henkin.

Let us fix $\Gamma$ to be a finitely satisfiable set of formulas. We begin by making an important observation about such sets, namely, it can always be extended by adding a formula or its negation, while preserving the property of finite satisfiability.

**Lemma 1.29** Let $\Gamma$ be finitely satisfiable, and let $\varphi$ be any formula. Then either $\Gamma \cup \{\varphi\}$ or $\Gamma \cup \{\neg \varphi\}$ is finitely satisfiable.

**Proof** Assume (for contradiction), neither $\Gamma \cup \{\varphi\}$ nor $\Gamma \cup \{\neg \varphi\}$ is finitely satisfiable. By definition of finite satisfiability, this means that there are finite subsets $\Gamma_0 \subseteq \Gamma \cup \{\varphi\}$ and $\Gamma_1 \subseteq \Gamma \cup \{\neg \varphi\}$ that are not satisfiable. Consider the (finite) set $\Delta = (\Gamma_0 \cup \Gamma_1) \setminus \{\varphi, \neg \varphi\}$. Observe that since $\Delta \subseteq \Gamma$, $\Delta$ is satisfiable. Let $v$ be a satisfying truth assignment for $\Delta$. Then either $v \models \varphi$ or $v \models \neg \varphi$. Therefore, either $v \models \Gamma_0$ or $v \models \Gamma_1$, which contradicts our assumption that both $\Gamma_0$ and $\Gamma_1$ are unsatisfiable. \[\square\]

The set of all formulas of propositional logic are countable, i.e., there is a 1-to-1, onto function $f : \mathbb{N} \to F$, where $F$ is the set of all propositional logic formulas. Therefore, we can enumerate all the formulas in propositional logic. Let $\varphi_0, \varphi_1, \ldots$ be an enumeration of all formulas. Let us, inductively, define a sequence of sets of formulas as follows.

\[
\begin{align*}
\Delta_0 &= \Gamma \\
\Delta_n &= \begin{cases} 
\Delta_{n-1} \cup \{\varphi_{n-1}\} & \text{if this is finitely satisfiable} \\
\Delta_{n-1} \cup \{\neg \varphi_{n-1}\} & \text{otherwise}
\end{cases}
\end{align*}
\]

Observe that the sequence is non-decreasing, i.e., for every $n$, $\Delta_n \subseteq \Delta_{n+1}$. Further, by induction on $n$, using Lemma 1.29, we can conclude that $\Delta_n$ is finitely satisfiable for all $n$. Finally, define

\[\Delta = \bigcup_{n \in \mathbb{N}} \Delta_n.\]

Since $\Delta_n$ is finitely satisfiable for all $n \in \mathbb{N}$, we can conclude that $\Delta$ is also finitely satisfiable.

**Proposition 1.30** $\Delta$ is finitely satisfiable.

**Proof** Consider any finite subset $X = \{\psi_1, \ldots, \psi_m\}$ of $\Delta$. Observe, by definition $\Delta$, for each $i$, there is some $n_i$ such that $\psi \in \Delta_{n_i}$. Taking $n = \max\{n_1, \ldots, n_m\}$, observe that $X \subseteq \Delta_n$. Since $\Delta_n$ is finitely satisfiable, $X$ is satisfiable. This means that $\Delta$ is finitely satisfiable. \[\square\]

Finite satisfiability of $\Delta$ implies that $\Delta$ is a complete set.

**Proposition 1.31** For any formula $\varphi$, $\neg \varphi \in \Delta$ if and only if $\varphi \notin \Delta$. 

Proof Without loss of generality assume that \( \varphi \) is the \( n \)th formula, i.e., \( \varphi = \varphi_n \). Now by definition, in step \( n \) of the construction of \( \Delta \), if \( \varphi \notin \Delta \) then \( \neg \varphi \in \Delta \). On the other hand, if \( \{ \varphi, \neg \varphi \} \subseteq \Delta \) then since \( \{ \varphi, \neg \varphi \} \) is not satisfiable, \( \Delta \) would not be finitely satisfiable. But since \( \Delta \) is finitely satisfiable, it must be the case that at most one out of \( \varphi \) and \( \neg \varphi \) belong to \( \Delta \). \( \square \)

We are now ready to complete the proof of Theorem 1.26. That is, we will show that \( \Gamma \) (which is finitely satisfiable) is satisfiable. Consider the truth assignment \( v \) defined as follows.

\[
v(p) = \begin{cases} T \text{ if } p \in \Delta \\ F \text{ if } \neg p \in \Delta \end{cases}
\]

Note that \( v \) is well-defined because by Proposition 1.31, for any proposition \( p \), exactly one among \( p \) and \( \neg p \) is in \( \Delta \). \( v \) shows that \( \Delta \) is satisfiable, because of the following result.

**Proposition 1.32** For any formula \( \varphi \in \Delta \), \( v \models \varphi \).

**Proof** Consider an arbitrary \( \varphi \in \Delta \). Let \( P = \text{occ}(\varphi) \) and \( P^\neg = \{ \neg p \mid p \in P \} \). Consider the set

\[
U = (\Delta \cap P) \cup (\Delta \cap P^\neg) \cup \{ \varphi \}.
\]

Since \( U \) is a finite subset of \( \Delta \), by Proposition 1.30, we have \( U \) is satisfiable. Let \( v' \) be a truth assignment such that \( v' \models U \). Observe that for every \( p \in \Delta \cap P \), \( v'(p) = T \) and for every \( \neg p \in \Delta \cap P^\neg \), \( v'(p) = F \). Therefore, \( v \) and \( v' \) agree on all propositions in \( P \). By Lemma 1.13, since \( v' \models \varphi \), we have \( v \models \varphi \). \( \square \)

Proposition 1.32 establishes the fact that \( v \models \Delta \). Since \( \Gamma \subseteq \Delta \), \( v \models \Gamma \). Therefore, \( \Gamma \) is satisfiable.

### 1.4.3 An Application of Compactness: Coloring Infinite Planar Graphs

In this section we present an application of the compactness theorem. We will show that all infinite planar graphs are 4-colorable. We begin by recalling the graph coloring problem, and its connection to propositional logic.

**Definition 1.33 (Graphs)**

An undirected graph \( G = (V,E) \) is a set of vertices \( V \), and a set of edges \( E \subseteq V \times V \), such that \( E \) is symmetric (i.e., \( (u,v) \in E \) iff \( (v,u) \in E \)) and irreflexive (i.e., \( (u,u) \notin E \) for any \( u \in V \)).

**Definition 1.34 (Coloring)**

A \( k \)-coloring of graph \( G = (V,E) \) is a function \( c : V \rightarrow \{1,2,\ldots,k\} \) such that if \( (u,v) \in E \) then \( c(u) \neq c(v) \). If \( G \) has a \( k \)-coloring then \( G \) is said to be \( k \)-colorable.

The problem of determining whether a graph is \( k \)-colorable can be “reduced” to checking the satisfiability of a set of formulas.
Proposition 1.35 For any graph \( G = (V, E) \) (with possibly infinitely many vertices), there is a set of formulas \( \Gamma_{G,k} \) such that \( G \) is \( k \)-colorable iff \( \Gamma_{G,k} \) is satisfiable.

Proof For each vertex \( u \in V \) and \( 1 \leq i \leq k \), take the proposition \( r_{ui} \) to denote “vertex \( u \) has color \( i \)”. \( \Gamma_{G,k} \) is the following set of formulas.

- For each \( u \in V \), the formula \( r_{u1} \lor r_{u2} \lor \cdots \lor r_{uk} \). Intuitively these formulas capture the constraint that every vertex gets at least one of the \( k \) colors.
- For each \( u \in V \) and \( 1 \leq i, j \leq k \) with \( i \neq j \), the formula \( \neg r_{ui} \lor \neg r_{uj} \). These formulas capture the constraint that a vertex does not get two different colors.
- For each edge \((u, v) \in E\) and color \( 1 \leq i \leq k \), the formula \( \neg r_{ui} \lor \neg r_{vi} \). These formulas ensure that adjacent vertices do not get the same color.

For a coloring \( c \), define the valuation \( v_c \) such that \( v_c(r_{ui}) = T \) iff \( c(u) = i \). Similarly for a valuation \( v \), define a function \( c_v(u) = i \) iff \( v(r_{ui}) = T \). Observe that

- If \( c \) is a valid \( k \)-coloring of \( G \) then \( v_c \) satisfies \( \Gamma_{G,k} \), and
- If \( v \) satisfies \( \Gamma_{G,k} \) then \( c_v \) is a valid \( k \)-coloring of \( G \).

Proof of the above observations is left as exercise. \( \Box \)

Finite, planar graphs are graphs with finitely many vertices such that there is a drawing of the graph on the plane where the edges do not cross. A celebrated result about finite, planar graphs is that 4 colors are sufficient to color the graph.

Theorem 1.36 (Appel-Haken)

Every finite planar graph is 4-colorable.

We will show that the compactness theorem in fact shows that Theorem 1.36 can be extended to infinite graphs as well.

Corollary 1.37 All infinite planar graphs are 4-colorable.

Proof Let \( G \) be an infinite planar graph. Consider the set of formulas \( \Gamma_{G,A} \) constructed in Proposition 1.35. Observe that \( \Gamma_{G,A} \) is finitely satisfiable. This can be seen as follows. Let \( \Gamma_0 \) be any finite subset of \( \Gamma_{G,A} \). Let \( G_0 \) be the graph induced by the vertices \( u \) such that the proposition \( p_{ui} \) appears in \( \Gamma_0 \) for some \( i \). Now, \( G_0 \) is a finite, planar graph and so by Theorem 1.36 has a 4-coloring \( c \). Then by the proof of Proposition 1.35, the valuation \( v_c \) satisfies \( \Gamma_{Gn,A} \). Since \( \Gamma_0 \subseteq \Gamma_{Gn,A} \), we have \( v_c \) satisfies \( \Gamma_0 \).

Since \( \Gamma_{G,A} \) is finitely satisfiable, by the compactness theorem, \( \Gamma_{G,A} \) is satisfiable. Let \( v \) be a satisfying assignment for \( \Gamma_{G,A} \). Again, by Proposition 1.35, \( c_v \) is a valid 4-coloring of \( G \). \( \Box \)
Chapter 2
Proof Systems

If logic is the science of valid inference, then proofs embody its heart. But what are mathematical proofs? They are a sequence of statements where each statement in the sequence is either a self evident truth, or “logically” follows from previous observations. Thus, sound derivation principles are identified by correct proofs.

Example 2.1 Euclid’s Elements sets out axioms (or postulates), which are self evident truths, and proves all results in geometry from these truths formally. Euclid lays out five axioms for geometry.

A1 A straight line can be drawn from any point to any point.
A2 A finite line segment can be extended to an infinite straight line.
A3 A circle can be drawn with any point as center and any given radius.
A4 All right angles are equal.
A5 If a straight line falling on two straight lines makes the interior angles on the same side less than two right angles, the straight lines, if produced indefinitely, will meet on that side on which the angles are less than two right angles.

Using these axioms, Euclid proves a number of results in geometry. He uses previously proved propositions in the proofs of later observations. An example of such a result, is the proof that the sum of the interior angles of a triangle is 180°.

Claim The interior angles of a triangle sum to two right angles.

Proof Consider the diagram in Fig. 2.1. The proposition is proved using the following sequence of statements.

1. Extend one side (say) BC to D [A2]
2. Draw a line parallel to AB through point C; call it CE [P31]
3. Since AB is parallel to CE, BAC = ACE and ABC = ECD [P29]
4. Thus, the sum of the interior angles = ACB + ACE + ECD = 180°

References [P31] and [P29] in steps 2 and 3, allude to previously propositions 31 and 29, proved in the book.

□
Example 2.1 highlights the basic elements of identifying good proofs — one needs to identify axioms, and the principle by which new conclusions can be drawn from previously established facts. A formal proof system for a logic identifies such axioms and rules of inference. We will introduce two such proof systems for propositional logic — a Frege-style proof system, and resolution — to give a flavor of different types of proof systems.

2.1 A Frege-style Proof System

Proof systems are most convenient presented as a collection of rules of the form

\[
\frac{\Gamma}{\varphi}
\]

where \( \Gamma \) is a set of formulas (schemas) and \( \varphi \) is a formula (schema). Such rules can be interpreted as follows — if every formula in \( \Gamma \) can be established then \( \varphi \) can be concluded from these observations. One special case is when \( \Gamma = \emptyset \). In this case the formula below the line can be concluded without establishing anything; in other words, it is an axiom. Instead of explicitly writing \( \emptyset \) above the line, we simply don’t write anything, and present this axiom in the form

\[
\overline{\varphi}
\]

Before presenting our first proof system, observe that all propositional logic formulas can be expressed using just implication and \( \bot \). To see this observe that \( \neg \varphi \) is the same as \( \varphi \to \bot \) and \( \varphi \lor \psi \) is \( (\varphi \to \bot) \to \psi \). Our first proof system, shown in Fig. 2.2, assumes that are formulas are written using implication and \( \bot \).

Our first proof system has 3 axiom schemes and one rule of inference. The formulas \( \varphi \), \( \psi \), and \( p \) in Fig. 2.2, can be any formulas. For example, taking \( \varphi = p \) and \( \psi = p \), we get \( p \to (p \to p) \) as an instantiation of the first axiom schema, while taking \( \varphi = p \) and \( \psi = p \to p \), we get \( p \to ((p \to p) \to p) \) as a different instantiation of the same schema. The rule of inference in this proof system, is a very commonly used rule. It, therefore, has a special name; it is called modus ponens.
2.1 A Frege-style Proof System

Fig. 2.2 A Frege-style Proof System

Proofs in our (formal) proof system, will be like the usual proofs in mathematics — they will be a sequence of statements. However, instead of using English statements, here they will simply be well formed formulas of propositional logic. The statement (or formula) being proved is the last one in the sequence. The sequence of formulas in a proof should be consistent with the axioms and rule of inference of the proof system, for it to be valid proof. This is captured in the definition below.

Definition 2.2 (Proofs)

A proof of $\phi$ from a set (possibly infinite) of hypotheses $\Gamma$ is a finite sequence of wffs $\psi_1, \psi_2, \ldots, \psi_m$ such that $\psi_m = \phi$, and for every $k \in \{1, 2, \ldots, m\}$, either

- $\psi_k \in \Gamma$, or
- $\psi_k$ is an axiom, or
- $\psi_k$ follows from $\psi_i$ and $\psi_j$, with $i, j < k$, by modus ponens.

The length of such a proof is the number of wffs in the sequence, namely, $m$. If there is a proof of $\phi$ from $\Gamma$, we denote this by $\Gamma \vdash \phi$. When $\Gamma = \emptyset$, we write this as $\vdash \phi$ (as opposed to $\emptyset \vdash \phi$).

Let us look at some proofs in our system.

Example 2.3 Let us construct a proof of $q \rightarrow p$ from the hypothesis $\{p\}$. Such a proof is as follows.

1. $p \rightarrow (q \rightarrow p)$ Axiom 1, taking $\phi = p$, and $\psi = q$
2. $p$ Hypothesis in set $\Gamma$
3. $q \rightarrow p$ Modus Ponens on lines 1 and 2

Thus, $\{p\} \vdash q \rightarrow p$.

We will now show that $\vdash \bot \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow p))$.

1. $(p \rightarrow (q \rightarrow p)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow p))$ Axiom 2, taking $\phi = p$, $\psi = q$ and $\rho = p$
2. $p \rightarrow (q \rightarrow p)$ Axiom 1, taking $\phi = p$, and $\psi = q$
3. $(p \rightarrow q) \rightarrow (p \rightarrow p)$ Modus ponens on lines 1 and 2
4. $((p \rightarrow q) \rightarrow (p \rightarrow p)) \rightarrow (\bot \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow p)))$ Axiom 1, taking $\phi = (p \rightarrow q) \rightarrow (p \rightarrow p)$, and $\psi = \bot$
5. $\bot \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow p))$ Modus ponens on lines 3 and 4
Finally, let us show $\vdash p \rightarrow p$.

1. $(p \rightarrow ((p \rightarrow p) \rightarrow p)) \rightarrow ((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p))$
   Axiom of $2$, taking $\varphi = p$, $\psi = p \rightarrow p$ and $\rho = p$
2. $(p \rightarrow ((p \rightarrow p) \rightarrow p))$
   Axiom $1$, taking $\varphi = p$, and $\psi = p \rightarrow p$
3. $(p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p)$
   Modus ponens on lines $1$ and $2$
4. $p \rightarrow (p \rightarrow p)$
   Axiom $1$, taking $\varphi = p$, $\psi = p$
5. $p \rightarrow p$
   Modus ponens on lines $3$ and $4$

In proof systems, like the one we are considering in this section, there is a very useful theorem that makes writing proofs easy. This is called the deduction theorem. Some proof systems have it as an explicit rule.

**Theorem 2.4 (Deduction Theorem)**

If $\Gamma \cup \{\varphi\} \vdash \psi$ then $\Gamma \vdash \varphi \rightarrow \psi$.

First, observe that the converse of Theorem 2.4, is clearly true, i.e., if $\Gamma \vdash \varphi \rightarrow \psi$ then $\Gamma \cup \{\varphi\} \vdash \psi$. Establishing this left as an exercise. The proof of the deduction theorem is a more difficult exercise. The informal outline of the proof is as follows. Assume that $\rho_1, \rho_2, \ldots, \rho_m$ is a proof of $\psi$ from $\Gamma \cup \{\varphi\}$. One shows by induction on $i$ that, for each line $i$, we have $\Gamma \vdash \varphi \rightarrow \rho_i$.

The deduction theorem simplifies the task of writing down proofs in our proof system.

**Example 2.5** Consider the task of showing $\vdash (\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \rho) \rightarrow (\varphi \rightarrow \rho))$, where $\varphi$, $\psi$, and $\rho$ are arbitrary wffs. Our approach to solving this problem, would instead be to instead establish $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \rho$. If we succeed, we will get the desired result by using the deduction theorem a few times.

1. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \varphi \rightarrow \psi$ Hypothesis
2. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \varphi$ Hypothesis
3. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \psi$ Modus Ponens on lines $1$ and $2$
4. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \psi \rightarrow \rho$ Hypothesis
5. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho, \varphi\} \vdash \rho$ Modus Ponens on lines $3$ and $4$
6. $\{\varphi \rightarrow \psi, \psi \rightarrow \rho\} \vdash \varphi \rightarrow \rho$ Deduction Theorem
7. $\{\varphi \rightarrow \psi\} \vdash (\psi \rightarrow \rho) \rightarrow (\varphi \rightarrow \rho)$ Deduction Theorem
8. $\vdash (\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \rho) \rightarrow (\varphi \rightarrow \rho))$ Deduction Theorem

### 2.1.1 Completeness Theorem

Proofs in a formal proof system like the one in Fig. 2.2 try capture the notion of correct logical inference. But what do we mean by “correct logical inference”? There
are two aspects to such a question. First, any conclusion drawn by a proof must be logically correct, i.e., consistent with the semantics we defined. In other words, we should not be able to conclude any false facts using a proof. This is often referred to as the soundness of the proof system. The second is that the proof system must be rich enough to be able to prove all true facts. This is called the completeness of the proof system. We make this connection between provability and the semantics we gave precise in the following theorem.

**Theorem 2.6 (Soundness and Completeness)**

For any set of formulas $\Gamma$ (possibly even infinite) and any wff $\varphi$ the following two properties hold.

**Soundness**  If $\Gamma \vdash \varphi$ then $\Gamma \models \varphi$.

**Completeness**  If $\Gamma \models \varphi$ then $\Gamma \vdash \varphi$.

Thus, any formula proved without any hypotheses is a tautology, and every tautology has a proof from the empty set of hypotheses in our proof system. We will not prove Theorem 2.6. We will instead prove such a soundness and completeness theorem for the second proof system that we will introduce for propositional logic. Proving soundness is usually easy. It requires making sure that the axioms and proof rules are consistent with the semantics of the logic. In this case it requires showing that every axiom in the proof system is indeed a tautology, and modus ponens is consistent with logical consequence. Proving completeness is typically hard.

### 2.2 Resolution

Notice that the proofs for formulas that we constructed in Examples 2.3 and 2.5 are very symbolic and mechanical — one doesn’t need to understand what the formula we are trying to prove is saying or what the meaning of the hypotheses is. Instead the proofs are constructed by looking at the pattern of formulas. This raises the prospect of trying to mechanize the process of searching for proofs of formulas. However, the proof system in Sect. 2.1 is not good for this purpose. This is because at any point during the construction of the proof, one can extend it by using any one of the axiom schemas. Since each axiom schema can be instantiated in infinitely many possible ways, this makes mechanization difficult. A proof system that is amenable to mechanization is one that has very few choices at any step of the proof construction. **Resolution** is such a proof system. Resolution has no axioms and only one rule of inference.

Resolution is a method for refutations, i.e., it proves that a formula is “not true” or more precisely unsatisfiable; recall that a formula $\varphi$ is unsatisfiable if $v \not\models \varphi$ for all valuations $v$. Thus unlike the proof system in Sect. 2.1 which proves validity of a formula directly, resolution works by showing that the negation of the formula is unsatisfiable. One can imagine resolution refutations like a proof by contradiction, where one assumes the negation of what one is trying to establish, and trying to show that that is impossible.
Resolution works when the formulas are represented in *conjunctive normal form* (CNF). We begin, therefore, by introducing conjunctive normal form formulas. CNF formulas are built using propositions and their negations, and the logical operators of $\land$ and $\lor$. It will be convenient to think of $\lor$ and $\land$ being present implicitly (see Definition 2.7 below) instead of explicitly in the syntax.

**Definition 2.7 (Conjunctive Normal Form)**

Conjunctive normal form formulas are defined as follows.

- A *literal* is a proposition $p$ or its negation $\neg p$.
- A *clause* is a disjunction of literals. We will think of a clause as a set of literals, implicitly assuming that the literals are disjuncted. In this interpretation, a truth valuation satisfies a clause if some literal in the set evaluates to $\text{true}$ under the truth valuation.
- The *empty clause* is the clause containing no literals. By definition, no truth assignment satisfies the empty clause.
- A formula is said to be in *conjunctive normal form* (CNF) if it is conjunction of clauses. Again, we will think of a CNF formula as a set of clauses, with the conjunction being implicit in the syntax. With this interpretation, a truth valuation satisfies a CNF formula if it satisfies every clause in the set representing the formula.

**CNF formulas as sets of sets of literals**

In Definition 2.7, clauses are represented as sets of literals, and CNF formulas as sets of clauses. Why is this well-defined? The reason is because $\lor$ and $\land$ are both *idempotent*, *commutative*, and *associative*. Idempotence means that disjuncting/conjuncting a formula with itself is equivalent to the formula. In other words, for any wff $\varphi$ we have

$$\varphi \lor \varphi \equiv \varphi \quad \varphi \land \varphi \equiv \varphi.$$  

Idempotence of disjunction and conjunction ensures that representing them as sets which don’t have repeated elements is faithful with the semantics. Commutativity and associativity mean that the order in which formulas are disjuncted/conjuncted does not change its semantics. That is, for any formulas $\varphi$, $\psi$, and $\rho$,

$$\varphi \lor \psi \equiv \psi \lor \varphi \quad \varphi \land \psi \equiv \psi \land \varphi$$

$$\varphi \lor (\psi \lor \rho) \equiv (\varphi \lor \psi) \lor \rho \quad (\varphi \land \psi) \land \rho \equiv \varphi \land (\psi \land \rho).$$

Thus, commutativity and associativity ensure that sets (which are ordered collections) are faithful representations of disjunctions and conjunctions of formulas.

CNF formulas are formulas in a restricted form. However, they are not semantically restrictive. That is, every wff $\varphi$ can be shown to be equivalent to a formula $\psi$ in CNF — we can push negations all the way in using DeMorgan’s Laws, and then
distribute \( \lor \) over \( \land \). We will look at this conversion process more closely later in this chapter. Let us look at some examples of CNF and non-CNF formulas.

**Example 2.8** The formulas \( (p_1 \land q_1) \lor (p_2 \land q_2) \), \( \neg(p \land q) \) are examples of formulas that are not in CNF — neither formula has \( \land \) as the topmost connective. The formulas \( (p_1 \lor p_2) \land (p_1 \lor q_2) \land (q_1 \lor p_2) \land (q_1 \lor q_2) \) and \( \neg p \lor \neg q \) are formulas in CNF as they are conjunctions of clauses.

We will represent CNF formulas as set of set of literals, without explicit conjunctions and disjunctions. For example, \( \{\{p_1, p_2\}, \{p_1, q_2\}, \{q_1, p_2\}, \{q_1, q_2\}\} \) is the way the formula \( (p_1 \lor p_2) \land (p_1 \lor q_2) \land (q_1 \lor p_2) \land (q_1 \lor q_2) \) will be represented. Similarly, \( \{\{\neg p, \neg q\}\} \) will be the representation of \( \neg p \lor \neg q \).

The resolution proof system is a sequence of transformations that preserve satisfiability, until the empty clause (which is by definition not satisfiable) is obtained. The transformations involve the single rule of inference that constructs the *resolvent* of two clauses.

**Definition 2.9 (Resolvent)** The only rule in the resolution proof system is as follows.

\[
\begin{align*}
C \cup \{p\} & \quad D \cup \{\neg p\} \\
\hline
C \cup D
\end{align*}
\]

The conclusion \( C \cup D \) is called the *resolvent* of \( C \cup \{p\} \) and \( D \cup \{\neg p\} \) with respect to proposition \( p \).

Two clauses may have multiple resolvents depending on which proposition one chooses to resolve with respect to. The resolvent of two clauses may be the empty clause if \( C \) and \( D \) are empty sets. Let us look at some examples to clarify this definition.

**Example 2.10** Consider the clauses \( \{p, \neg q, \neg r\} \) and the clause \( \{\neg p, \neg q\} \). The resolvent of these two clauses (with respect to \( p \)) is the clause \( \{\neg q, \neg r\} \).

On the other hand, if we consider clauses \( \{p, \neg q\} \) and \( \{\neg p, q\} \), we have two possible resolvents. If we resolve with respect to \( p \), we get \( \{q, \neg q\} \), and if we resolve with respect to \( q \), we get \( \{p, \neg p\} \).

**Definition 2.11 (Refutations)**

A *resolution refutation* of a (possibly infinite) set of clauses \( \Gamma \) is a sequence of clauses \( C_1, C_2, \ldots, C_m \) such that each clause \( C_k \) is either in \( \Gamma \) or a resolvent of two clauses \( C_i \) and \( C_j \) \((i, j < k)\), and the last clause \( C_m \) in the refutation is the empty clause.

**Example 2.12** The set of clauses \( \Gamma = \{\{p, q\}, \{\neg p, r\}, \{\neg q, r\}, \{\neg r\}\} \) has the following resolution refutation.
1. \{\neg p, r\}
2. \{\neg r\}
3. \{\neg p\}  \quad \text{Resolvent of 1 and 2}
4. \{\neg q, r\}
5. \{\neg q\}  \quad \text{Resolvent of 2 and 4}
6. \{p, q\}
7. \{q\}  \quad \text{Resolvent of 3 and 6}
8. \{\}\  \quad \text{Resolvent of 5,7}

### 2.2.1 Proving Tautologies with Resolution

As mentioned before, resolution refutations establish the unsatisfiability of a formula given in CNF. To prove that a formula is a tautology using resolution, we need to use Proposition 1.24. That is, to prove that \( \varphi \) is a tautology, we need to convert \( \neg \varphi \) into CNF. This can be done as follows.

1. Push negations inside using DeMorgan’s Laws. Recall that DeMorgan’s laws say the following.

\[
\neg(\psi_1 \land \psi_2) \equiv \neg\psi_1 \lor \neg\psi_2 \quad \neg(\psi_1 \lor \psi_2) \equiv \neg\psi_1 \land \neg\psi_2
\]

2. Remove double negations, because \( \neg \neg \psi \equiv \psi \)

3. Distribute \( \lor \) over \( \land \), using the distributive law, which says

\[
\psi_1 \lor (\psi_2 \land \psi_3) \equiv (\psi_1 \lor \psi_2) \land (\psi_1 \lor \psi_3) \quad (\psi_1 \land \psi_2) \lor \psi_3 \equiv (\psi_1 \lor \psi_3) \land (\psi_2 \lor \psi_3)
\]

**Example 2.13** Let \( \varphi = (\neg p_1 \lor \neg q_1) \land (\neg p_2 \lor \neg q_2) \). We can convert \( \neg \varphi \) to CNF as follows.

1. Pushing negations inside using DeMorgan’s Laws, we get

\[
(\neg\neg p_1 \land \neg\neg q_1) \lor (\neg\neg p_2 \land \neg\neg q_2)
\]

2. Removing double negations, we get

\[
(p_1 \land q_1) \lor (p_2 \land q_2)
\]

3. Distributing \( \lor \) over \( \land \), we get

\[
(p_1 \lor p_2) \land (p_1 \lor q_2) \land (q_1 \lor p_2) \land (q_1 \lor q_2)
\]

The above method while semantically correct, can be expensive. The main reason is that when we distribute \( \lor \) over \( \land \), the resulting formula can be exponentially larger.

For example, if we have the formula...
and we distribute \( \lor \) over \( \land \), we will get a CNF formula where each clause is of the form \((r_1 \lor r_2 \lor \cdots \lor r_n)\), where \( r_i \) is either \( p_i \) or \( q_i \). This will result in a formula with \( 2^n \) clauses (as we have two choices for each \( r_i \)).

The exponential blowup can be avoided by using a translation proposed by Tseitin. Tseitin’s method does not construct a logically equivalent CNF formula. Instead, for the formula \( \neg \varphi \), it constructs a CNF formula \( \psi \) with the property that \( \neg \varphi \) is satisfiable if and only if \( \psi \) is satisfiable. This weaker correspondence between \( \neg \varphi \) and \( \psi \) is sufficient in this context; we will have \( \varphi \) is a tautology if and only if \( \neg \psi \) is unsatisfiable.

Tseitin’s Method.

Let us describe the conversion of formula \( \neg \varphi \). The first step is to introduce new extension propositions \( x_\psi \) for each subformula \( \psi \) of \( \varphi \) as follows.

- For each proposition in \( \varphi \), the extension proposition \( x_\psi \) is the same as \( \psi \).
- For each negated subformula \( \neg \psi \), \( x_{\neg \psi} \) is taken to be the literal \( \neg x_\psi \).
- For all other subformulas \( \psi \), \( x_\psi \) is a new proposition.

Having identified the extension propositions, the CNF formula that we will construct corresponding to \( \neg \varphi \) is as follows. Here, we will use the representation of CNF formulas as sets of sets of literals. So for \( \varphi \), we define \( \Gamma_\varphi \) to be the following set of clauses.

- The singleton clause \( \{ \neg x_\varphi \} \)
- For each subformula \( \psi \land \rho \), we add the CNF formula equivalent to \( \neg x_{\psi \land \rho} \leftrightarrow x_\psi \land x_\rho \). In other words, we will have the clauses
  \[
  \{ \neg x_{\psi \land \rho}, x_\psi \} \quad \{ \neg x_{\psi \land \rho}, x_\rho \} \quad \{ x_{\psi \land \rho}, \neg x_\psi, \neg x_\rho \}
  \]
- For each subformula \( \psi \lor \rho \), we add the CNF formula equivalent to \( \neg x_{\psi \lor \rho} \leftrightarrow x_\psi \lor x_\rho \). In other words, we will have the clauses
  \[
  \{ x_{\psi \lor \rho}, \neg x_\psi \} \quad \{ x_{\psi \lor \rho}, \neg x_\rho \} \quad \{ \neg x_{\psi \lor \rho}, x_\psi, x_\rho \}
  \]

Observe that the number of subformulas of a given formula \( \varphi \) is linear in the size of \( \varphi \). Thus the number of extension propositions we introduce is linear in \( \varphi \). Further, for each subformula, we are introducing only a constant (3 to be precise) number of clauses. This means that resulting set of clauses \( \Gamma_\varphi \) is linear in the size of \( \varphi \).

**Example 2.14** Let us apply Tseitin’s construction to the formula in Example 2.13.

Recall that \( \varphi = (\neg p_1 \lor \neg q_1) \land (\neg p_2 \lor \neg q_2) \). The first step is to identify the new extension propositions we need. In this case there are only 3 that we will add — \( x_\varphi \) corresponding to \( \varphi \), \( x_1 \) corresponding to \( (\neg p_1 \lor \neg q_1) \), and \( x_2 \) corresponding to...
\[ (-\neg p_2 \lor \neg q_2). \] Our CNF formula \( \Gamma_\varphi \) will be obtained by adding 3 clauses for each of these interesting subformulas corresponding to \( x_\varphi, x_1 \) and \( x_2 \). Thus, we have \( \Gamma_\varphi \) is the following set

\[
\begin{align*}
\{ \neg x_\varphi \}, \\
\{ \neg x_\varphi, x_1 \}, \\
\{ \neg x_\varphi, x_2 \}, \\
\{ x_\varphi, \neg x_1, \neg x_2 \}, \\
\{ x_1, p_1 \}, \\
\{ x_1, q_1 \}, \\
\{ x_2, p_2 \}, \\
\{ x_2, q_2 \}, \\
\end{align*}
\]

In the above description, we have replaced \( \neg \neg p \) by \( p \) for \( p \in \{ p_1, q_1, p_2, q_2 \} \).

2.2.2 Completeness of Resolution

We will now prove that resolution is a “correct” proof system. In other words, we will prove the soundness and completeness of resolution. What that means for resolution is that a set of clauses \( \Gamma \) has a resolution refutation if and only if \( \Gamma \) is unsatisfiable. Note that we will establish this for any set \( \Gamma \), including those that are infinite. Recall that a set of clauses \( \Gamma \) is satisfiable if there is a truth assignment \( v \) such that for every clause \( C \in \Gamma \), there is some literal \( \ell \in C \) such that \( v(\ell) = T \).

**Theorem 2.15 (Soundness)**

If a set of clauses \( \Gamma \) has a resolution refutation, then \( \Gamma \) is unsatisfiable.

**Proof** The crux of the proof of the soundness theorem, is to establish the “correctness” of the resolution proof rule. That is captured by the following lemma.

**Lemma 2.16 (Resolution Lemma)**

Let \( \Delta \) be a set of clauses and let \( C \) be the resolvent of two clauses \( D, E \in \Delta \). Then for any assignment \( v \), \( v \models \Delta \) if and only if \( v \models \Delta \cup \{ C \} \).

**Proof (Of Lemma 2.16)** Observe that if \( v \models \Delta \cup \{ C \} \), then clearly \( v \models \Delta \).

Consider a truth assignment such that \( v \models \Delta \). Without loss of generality, let us assume there is a proposition \( p \) such that \( p \in D \) and \( \neg p \in E \). If \( v(p) = T \) then since \( v \models E \), there must be a literal \( \ell \in E \) (obviously \( \ell \neq \neg p \)) such that \( v(\ell) = T \). Since \( \ell \in C \), we have \( v \models C \). On the other hand, if \( v(p) = F \) then since \( v \models D \), there must be a literal \( \ell \in D \) (obviously \( \ell \neq p \)) such that \( v(\ell) = T \). Since \( \ell \in C \), we have \( v \models C \).

Using Lemma 2.16, we are ready to complete the proof of Theorem 2.15. Let \( C_1, C_2, \ldots, C_m \) be a resolution refutation of \( \Gamma \). Let us define a sequence of sets of clauses inductively as follows.

\[
\Gamma_0 = \Gamma \quad \Gamma_i = \Gamma_{i-1} \cup \{ C_i \}
\]

Thus, \( \Gamma_i = \Gamma \cup \{ C_1, C_2, \ldots, C_i \} \). Since \( C_m = \{ \} \), \( \Gamma_m \) is clearly unsatisfiable. Therefore, by Lemma 2.16 (and induction), \( \Gamma = \Gamma_0 \) is also unsatisfiable.
Theorem 2.17 (Completeness)

*If a set of clauses \( \Gamma \) is unsatisfiable then there is a resolution refutation of \( \Gamma \).*

We will present a proof of the completeness theorem due to David and Putnam. For finite \( \Gamma \), the proof is constructive. That is, when \( \Gamma \) is unsatisfiable, it gives a specific construction of a refutation for \( \Gamma \).

**Proof** Let \( \Gamma \) be an unsatisfiable set of clauses. By the compactness theorem (Theorem 1.26), there is a finite subset \( \Delta \subseteq \Gamma \) that is unsatisfiable; this can be seen by taking \( \phi = \bot \) in Corollary 1.27. We will prove the completeness theorem by induction on the number of propositions appearing in \( \Delta \). Before outlining the proof, it is useful to point out that since \( \Delta \) is unsatisfiable, it must be a non-empty set of clauses. This is because an empty set of clauses is, by definition, satisfiable.

For the base case, observe that if \( \Delta \) contains no propositions, then \( \Delta \) contains the empty clause. Then the refutation for \( \Delta \) is simply \( \{\}\).

Let us now consider the induction step. Let us call a clause \( C \) trivial if there is a proposition \( ? \) such that \( \{?, \neg ?\} \subseteq C \). Trivial clauses are valid, and hence they can be removed from \( \Delta \) without affecting its satisfiability. Thus, without loss of generality, we will assume that \( \Delta \) does not have any trivial clauses.

Let \( ? \) be a proposition that appears in \( \Delta \) with respect to proposition \( ? \). \( \Delta \) can be partitioned into 3 sets.

\[
\begin{align*}
\Delta_0^p &= \{ C \in \Delta \mid C \cap \{p, \neg p\} = \emptyset \} \\
\Delta_p &= \{ C \in \Delta \mid p \in C \} \\
\Delta_\neg p &= \{ C \in \Delta \mid \neg p \in C \}
\end{align*}
\]

Thus, \( \Delta_0^p \) are clauses where \( p \) does not appear, \( \Delta_p \) are those where \( p \) appears positively, and \( \Delta_\neg p \) are those where \( p \) appears negatively. Let us construct a new set of clauses as follows.

\[
\Delta_p = \Delta_0^p \cup \{ C \cup D \mid C \cup \{p\} \in \Delta_p \text{ and } D \cup \{\neg p\} \in \Delta_\neg p \}
\]

Thus, \( \Delta_p \) has all the clauses in \( \Delta_0^p \) and all resolvents of clauses from \( \Delta_p \) and \( \Delta_\neg p \). Observe that \( p \) no longer appears in \( \Delta_p \). If we can argue that \( \Delta_p \) is unsatisfiable then we can complete the proof by using the induction hypothesis — the refutation for \( \Delta \) is just all the steps to create \( \Delta_p \) followed by a refutation for \( \Delta_p \).

To finish the proof, we need to establish the following lemma.

**Lemma 2.18** If \( \Delta_p \) is satisfiable then so is \( \Delta \). \( \square \)

**Proof (Of Lemma 2.18)** Let \( v \) be a truth assignment that satisfies \( \Delta_p \). Let \( v' \) be the truth assignment that is identical to \( v \), except that it flips the assignment to \( p \). Observe that since \( v \) and \( v' \) only differ on the assignment to \( p \), they agree on all the propositions appearing in \( \Delta_p \). Therefore, \( v' \) also satisfies \( \Delta_p \).

Let us assume without loss of generality, that \( v(p) = T \) and \( v'(p) = F \). We will show that either \( v \) or \( v' \) satisfies \( \Delta \). Observe that both \( v \) and \( v' \) satisfy \( \Delta_0^p \) (because \( p \) does not appear in \( \Delta_0^p \)). Also, \( v \) satisfies \( \Delta_p \) (because all clauses in \( \Delta_p \) have \( p \)) and \( v' \) satisfies \( \Delta_\neg p \) (because all clauses in \( \Delta_\neg p \) have \( \neg p \)). Now if \( v \) satisfies \( \Delta_p \), \( v \) satisfies...
Similarly, if $v'$ satisfies $\Delta_p^+\setminus\{p\}$ then $v'$ satisfies $\Delta$. So the problem is if neither of these hold. In that case that is a clause $C \cup \{p\} \in \Delta_p^+$ that is not satisfied by $v'$ and there is a clause $D \cup \{\neg p\} \in \Delta_p^+$ that is not satisfied by $v$. But then their resolvent $C \cup D \in \Delta_p$ is not satisfied by either $v$ or $v'$, which contradicts our assumption that both $v$ and $v'$ satisfy $\Delta_p$.

With the proof of Lemma 2.18, we have completed the proof of Theorem 2.17. □

### 2.3 Craig’s Interpolation Theorem and Proof Complexity

Craig’s Interpolation theorem is a classical result in logic that holds for many different logics. The theorem has been used in different contexts in the broad area of formal methods and verification — in hardware and software specification; reasoning about large knowledge databases; type inference; combination of theorem provers for different first order theories; model checking of finite and infinite state systems through the construction of abstractions. In this section we look at some of its connections to theoretical computer science and complexity theory in particular. We will introduce the theorem for propositional logic, and its connection with proof lengths for propositional logic formulas.

#### 2.3.1 Craig’s Interpolation Theorem

Before we state and prove the interpolation theorem, it will be convenient to introduce some notation. A list of propositions $p_1, p_2, \ldots, p_n$ will be denoted by $\overline{p}$ when the actual number of propositions in the list is unimportant. A formula $\varphi$ over propositions $p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_m$ will be sometimes denoted as $\varphi(\overline{p}, \overline{q})$, making explicit the propositions that may syntactically appear in $\varphi$. Finally, for a truth valuation $v$ and a list of propositions $\overline{p}$, $v \models \overline{p}$ will denote the restriction of $v$ to the propositions $\overline{p}$; note that $v$ has an infinite domain, the domain of $v \models \overline{p}$ is finite and restricted to $\overline{p}$.

**Theorem 2.19 (Craig)**

Suppose $\models \varphi(\overline{p}, \overline{q}) \Rightarrow \psi(\overline{q}, \overline{p})$. Then there is a formula $\eta(\overline{q})$ such that $\models \varphi(\overline{p}, \overline{q}) \Rightarrow \eta(\overline{q})$ and $\models \eta(\overline{q}) \Rightarrow \psi(\overline{q}, \overline{p})$. $\eta$ is said to be the interpolant of $\varphi$ and $\psi$.

Before presenting the proof of Theorem 2.19, let us examine its statement. There are different equivalent ways of presenting this result. Recall that $\models \varphi \Rightarrow \psi$ if $\models \varphi$. Therefore, we could say that if $\psi$ is a logical consequence of $\varphi$ then $\eta$ is a formula that captures the reason why, using only the common propositions. In this case, since $\varphi \models \eta$, we could think of $\eta$ as an “abstraction” of $\varphi$ (as $\eta$ “forgets” the constraints $\varphi$ imposes on $\overline{p}$) that is sufficient to ensure $\psi$. Another formulation of Craig’s theorem
Craig’s Interpolation Theorem and Proof Complexity

is as follows. Suppose \( \varphi(\vec{p}, \vec{q}) \land \psi(\vec{q}, \vec{r}) \) is unsatisfiable (or equivalently, \( \varphi \models \neg \psi \)), then there is a formula \( \eta(\vec{q}) \) over the common propositions such that \( \varphi \models \eta \) and \( \eta \land \psi \) is unsatisfiable (i.e., \( \eta \models \neg \psi \)). Informally, \( \eta \) is an abstraction of \( \varphi \) that captures why \( \varphi \land \psi \) is unsatisfiable. We will consider this formulation of Craig’s theorem in terms of unsatisfiability, when we revisit resolution later in this section.

Proof (Of Theorem 2.19) The proof of Craig’s Interpolation Theorem is quite simple in this context of propositional logic. Let us define

\[
M = \{ v \models \eta \mid v[\varphi] = T \}.
\]

Observe that since the domain (and range) of all functions in \( M \) is finite, \( M \) is a finite set. Let us, without loss of generality, take \( M = \{ v_1, v_2, \ldots, v_n \} \). The interpolant \( \eta \) will essentially say that “one among the assignments in \( M \) hold”. Formally,

\[
\eta(\vec{q}) = \lor_{i=1}^{n} (q_1^{(i)} \land q_2^{(i)} \land \cdots \land q_k^{(i)})
\]

where

\[
q_j^{(i)} = \begin{cases} q_j & \text{if } v_i(q_j) = T \\ \neg q_j & \text{otherwise} \end{cases} \quad \text{and} \quad \vec{q} = q_1, \ldots, q_k
\]

Clearly from the definition of \( M \) and \( \eta \), we have \( \varphi(\vec{p}, \vec{q}) \models \eta(\vec{q}) \). Let us now argue that \( \eta(\vec{q}) \models \psi(\vec{q}, \vec{r}) \). Consider an arbitrary valuation \( v \) such that \( v[\varphi] = F \). Since we have \( \varphi \models \psi \), it must be that \( v[\psi] = F \). Consider any assignment \( v' \) such that \( v \models \neg \psi \models v' \models \neg \psi \). Since \( v \) and \( v' \) agree on the propositions appearing in \( \psi \), we have \( v'[\psi] = F \). Again, since \( \psi \) is a logical consequence of \( \varphi \), it must be that \( v'[\varphi] = F \). Thus, no matter how the assignment to propositions \( \vec{p} \) is changed from \( v \), we will not be able to satisfy \( \varphi \). This means that \( v \models \neg \psi \not\in M \). Thus, by our construction of \( \eta \), \( v[\eta] = F \). This establishes that \( \eta \models \psi \).

Example 2.20 Let us look at a simple example that illustrates the construction of the interpolant in the proof of Theorem 2.19. Consider \( \varphi = (p \land (q_1 \lor q_2)) \) and let \( \psi = (q_1 \lor q_2 \lor r) \). It is easy to see that \( \models \varphi \rightarrow \psi \). The set \( M \) constructed in the proof in this case would be \( M = \{ v_{i1}, v_{i2}, assign_{i1} \} \), where \( v_{ij} \) is the function

\[
v_{ij}(q_1) = i \quad \text{and} \quad v_{ij}(q_2) = j
\]

Given \( M \), the proof constructs the follow formula as interpolant.

\[
\eta = (q_1 \land q_2) \lor (q_1 \land \neg q_2) \lor (\neg q_1 \land q_2).
\]

2.3.2 Size of Interpolants

The interpolant \( \eta \) constructed in the proof of Theorem 2.19 is exponential in the number of common variables, and therefore, could be exponential in the size of the formulas \( \varphi \) and \( \psi \). Can this be improved? Small interpolants can have a big impact in
the contexts where interpolants are used, like in formal verification. Or can we prove that, in the worst case, the interpolant needs to be exponential in the size of the input formulas? Unfortunately, like many questions in theoretical computer science, this remains open and unresolved — we cannot prove or disprove that the construction in the proof of Theorem 2.19 is the best. However, in this section, we will show that it is unlikely that we can construct polynomial sized interpolants for all formulas. Or more precisely, we will show that resolving whether there exist polynomial sized interpolants for all formulas, is closely related to other open questions in complexity theory.

In order to present these results on the size of interpolants, we need to introduce new circuit complexity classes. Circuit complexity for a problem is based on a *non-uniform* model of computation. The idea is the following. Imagine you have the ability to choose a different algorithm for each input length; how much resource would you need? The “programs” for each input length are circuits, and different aspects of these circuits correspond to different computational resources one may care about. Running time in this context, roughly corresponds to circuit size. This leads us to analogs of \( P, \) \( NP, \) \( \text{coNP} \) in the context of non-uniform complexity, which are defined next.

**Definition 2.21 (Circuits)**

A *Boolean Circuit* \( C \) is a sequence of assignments \( A_1, A_2, \ldots A_n, \) where each \( A_i \) is one of the following forms.

\[
\begin{align*}
P_1 &= F \\
P_1 &= T \\
P_1 &= ? \\
P_1 &= P_j \land P_k, \ j, k < i \\
P_1 &= P_j \lor P_k, \ j, k < i \\
P_1 &= \neg P_j, \ j < i 
\end{align*}
\]

where each \( P_i \) is a variable that appears on the left-hand side in only \( A_i. \) The size of such a circuit, denoted \( |C|, \) is \( n. \)

The variable \( P_i \) is said to be an *input variable* if the line \( A_i \) corresponding to it is of the form \( P_i = ?. \) The input variables of \( C \) will be denoted by \( I(C). \) Given an assignment \( \nu : I(C) \rightarrow \{T, F\}, \) the value of \( C \) under \( \nu \) is the value assigned to the variable \( P_n \) in the last line. There is a natural order on the variables (based on which line they are assigned a value) which also imposes an order on the input variables. Thus, an assignment \( I(C) \rightarrow \{T, F\} \) can be thought of as a string \( x, \) where \( x[i] \) is the value assigned to the \( i \)th input variable. In such a situation, the value of \( C \) under string \( x, \) will be denoted by \( C(x). \)

**Example 2.22** Circuits and formulas are two ways to represent Boolean functions. It is instructive to see how they differ by looking at an example.

Consider the boolean function \( \text{parity}(x_1, x_2, \ldots x_n) \) which computes whether the number of propositions in \( \{x_1, x_2, \ldots x_n\} \) set to \( T \) is odd or even. For example, we could write down the formula for some simple cases.
2.3 Craig’s Interpolation Theorem and Proof Complexity

\[
\text{parity}(x_1, x_2) = (x_1 \lor x_2) \land \neg(x_1 \land x_2)
\]
\[
\text{parity}(x_1, x_2, x_3) = (((x_1 \lor x_2) \land \neg(x_1 \land x_2)) \lor x_3) \\
\land \neg(((x_1 \lor x_2) \land \neg(x_1 \land x_2)) \land x_3)
\]

More generally, we can inductively write down the formula for \text{parity}(x_1, x_2, \ldots, x_n) by observing that the parity of \(x_1, x_2, \ldots, x_n\) is odd iff either (a) the parity of \(x_1, \ldots, x_{n-1}\) is even and \(x_n\) is \(T\), or (b) parity of \(x_1, \ldots, x_{n-1}\) is odd and \(x_n\) is \(F\). This results in the following definition:

\[
\text{parity}(x_1, x_2, \ldots, x_{n-1}, x_n) = (\text{parity}(x_1, x_2, \ldots, x_{n-1}) \lor x_n) \\
\land \neg(\text{parity}(x_1, x_2, \ldots, x_{n-1}) \land x_n)
\]

Observe that the formula we wrote down for \text{parity}(x_1, x_2, x_3) is based on exactly this definition. Observe that the size of the formula \text{parity}(x_1, \ldots, x_n) is double the size of \text{parity}(x_1, \ldots, x_{n-1}). This means that the size of parity for \(n\) arguments is \(O(2^n)\).

A circuit for the same formula does not grow as rapidly. This is because it can "reuse" previous computed answers. Let us look at the circuit corresponding to the formula \text{parity}(x_1, x_2, x_3) we wrote above.

\[
\begin{align*}
x_1 &= ? \\
x_2 &= ? \\
x_3 &= ? \\
P_4 &= x_1 \lor x_2 \\
P_5 &= x_1 \land x_2 \\
P_6 &= \neg P_5 \\
P_7 &= P_4 \land P_6 \\
P_8 &= P_7 \lor x_3 \\
P_9 &= P_7 \land x_3 \\
P_{10} &= \neg P_9 \\
P_{11} &= P_8 \land P_{10}
\end{align*}
\]

Notice that variable \(P_7\) stores parity \(x_1, x_2\) and it is reused without paying an extra cost. More generally, if \(C_{n-1}\) is the circuit computing \text{parity}(x_1, \ldots, x_{n-1}) with last line \(P_{k_{n-1}}\), the circuit \(C_n\) computing \text{parity}(x_1, \ldots, x_n) is given by

\[
\begin{align*}
C_{n-1} \\
x_n &= ? \\
Q_1 &= P_{k_{n-1}} \lor x_n \\
Q_2 &= P_{k_{n-1}} \land x_n \\
Q_3 &= \neg Q_2 \\
P_{k_n} &= Q_1 \land Q_3
\end{align*}
\]

Now if \(|C_n| = |C_{n-1}| + 5\). Therefore, in general, we get \(|C_n| = O(n)\). Thus the circuit for the same syntactic formula can be exponentially smaller.
Definition 2.23 A language $A \subseteq \{0, 1\}^*$ is said to be in $P/poly$ iff there are constants $c, k \in \mathbb{N}$, and a (infinite) family of circuits $\{C_i\}_{i \in \mathbb{N}}$ such that (a) for every $n$, $|C_n| \leq cn^k$, and (b) for every $x, y \in A$ iff $C_{|x|}(x) = T$.

A language $A \subseteq \{0, 1\}^*$ is said to be in $NP/poly$ (or $coNP/poly$) iff there are constants $c, k \in \mathbb{N}$, and a (infinite) family of circuits $\{C_i\}_{i \in \mathbb{N}}$ such that (a) for every $n$, $|C_n| \leq cn^k$, and (b) for every $x, y \in A$ iff for some (all) $r$, $C_{|x|}(x, r) = T$.

One can think of $P/poly$ (or $NP/poly$ or $coNP/poly$) as the collection of problems that be solved in polynomial time (or nondeterministic polynomial time or co-nondeterministic polynomial time) given a polynomially long advice string, namely, the description of the appropriate circuit. Just like in the case of (uniform/regular) complexity classes where it is open whether $P \subseteq NP \cap coNP$, the same question is open in the non-uniform context as well. That is, a long standing open problem is whether $P/poly \subseteq NP/poly \cap coNP/poly$.

Mundici’s theorem says that establishing the existence of interpolants with polynomial circuit representation for all pairs of formulas, is equivalent to resolving the $P/poly$ versus $NP/poly \cap coNP/poly$ question. Thus, it is likely to be difficult to establish.

Theorem 2.24 (Mundici)

If for any $\varphi$ and $\psi$ such that $\varphi \models \psi$ there is an interpolant whose circuit size is polynomial in $\varphi$ and $\psi$ then

$$P/poly = NP/poly \cap coNP/poly$$

Proof Consider a problem $L \in NP/poly \cap coNP/poly$. Thus, there are families of circuits $\{A_n\}_{n \in \mathbb{N}}$ and $\{B_n\}_{n \in \mathbb{N}}$ such that for each $n$, $A_n$ and $B_n$ have size bounded by a polynomial function of $n$ and for any binary string $w, w \in L$ iff $\exists p. A_{|w|}(p, w) = T$ iff $\forall r. B_{|w|}(w, r) = T$. These observations are just a consequence of $L \in NP/poly$ and $L \in coNP/poly$.

Based on the previous paragraph, we have, for any $n$, if for some assignment of values to $\overrightarrow{q}$, if $\exists \overrightarrow{p}. A_n(\overrightarrow{p}, \overrightarrow{q})$ holds then it also the case that $\forall \overrightarrow{r}. B_n(\overrightarrow{q}, \overrightarrow{r})$ holds. Therefore, we have $A_n(\overrightarrow{p}, \overrightarrow{q}) \models B_n(\overrightarrow{q}, \overrightarrow{r})$. By our assumption on interpolants, we have an interpolant (as a circuit) $C_n(\overrightarrow{q})$ such that $|C_n|$ is bounded by a polynomial in $|A_n|$ and $|B_n|$. Since $|A_n|$ and $|B_n|$ are bounded by polynomials in $n$, we have $|C_n|$ is bounded by a polynomial in $n$. Further, since $C_n$ is an interpolant, we have if $\exists \overrightarrow{p}. A_n(\overrightarrow{p}, \overrightarrow{q})$ holds then $C_n(\overrightarrow{q})$ holds, and if $C_n(\overrightarrow{q})$ holds then $\forall \overrightarrow{r}. B_n(\overrightarrow{q}, \overrightarrow{r})$ holds. Thus, $\{C_n\}$ is a family of polynomially sized circuits deciding $L$, and thereby demonstrating that $L \in P/poly$. $\square$

2.3.3 Interpolants from Refutations

Constructing small interpolants for all formulas is likely to be difficult. However, it turns out that, if a formula $A(\overrightarrow{p}, \overrightarrow{q}) \rightarrow B(\overrightarrow{q}, \overrightarrow{r})$ has a short proof in some
proof systems, then the proof can be used to construct an interpolant, whose size is propositional to the original proof. One such proof system that admits such a result is resolution.

**Theorem 2.25** Let the collection of clauses \( \Gamma = \{ A_i(\overline{P}, \overline{Q})\}_{i=1}^k \cup \{ B_j(\overline{Q}, \overline{P})\}_{j=1}^r \) have a resolution refutation of length \( n \). Then there is a circuit \( C(\overline{Q}) \) such that

\[
\bigwedge_i A_i(\overline{P}, \overline{Q}) \models C(\overline{Q}) \text{ and } C(\overline{Q}) \land \bigwedge_j B_j(\overline{Q}, \overline{P}) \text{ is unsatisfiable.}
\]

Further \( |C| \) is \( O(n) \).

**Proof** Since \( \Gamma = \{ A_i(\overline{P}, \overline{Q})\}_{i=1}^k \cup \{ B_j(\overline{Q}, \overline{P})\}_{j=1}^r \) is unsatisfiable, every truth valuation \( v \) fails to satisfy at least one of the \( A_i(\overline{P}, \overline{Q}) \) or \( B_j(\overline{Q}, \overline{P}) \). We can also restate the properties of an interpolant \( C \) as

\[
\models \neg C(\overline{Q}) \rightarrow \neg \bigwedge_i A_i(\overline{P}, \overline{Q}) \text{ and } \\
\models C(\overline{Q}) \rightarrow \neg \bigwedge_j B_j(\overline{Q}, \overline{P})
\]

Thus, \( C(\overline{Q}) \) can be thought of as a way of labelling truth valuations: those labelled \( F \) will not satisfy some clause \( A_i(\overline{P}, \overline{Q}) \) and those labelled \( T \) will not satisfy some \( B_j(\overline{Q}, \overline{P}) \).

The structure of the proof will be as follows. Let \( \psi_1, \psi_2, \ldots, \psi_n \) be a resolution refutation of \( \Gamma \). Let the set of common variables \( \overline{Q} = \{ q_1, \ldots, q_m \} \). Our circuit for the interpolant will be a sequence of the form \( q_1 = ?, q_2 = ?, \ldots, q_m = ?, P_1 = E_1, P_2 = E_2, \ldots, P_n = E_n \) — the first \( m \) lines asserting that \( q_i \)'s are variables, and then having one line \( P_i = E_i \) corresponding to each line \( \psi_i \) of our refutation. It will be convenient to consider expressions \( E_i \) on the right hand side that have more than one logical connective. We will find it convenient to talk about the “circuit corresponding to a line \( \psi_i \) in the refutation”. What we mean by this is look at the sequence of assignments upto (and including) line \( P_i = E_i \); we will denote this as circuit \( C_i \). The value of \( C_i \) (with respect to a truth assignment) will simply be the value variable \( P_i \) gets when we compute this circuit.

As mentioned above, we will construct the circuit line by line, corresponding to the refutation. With each line \( \psi_i \) in the refutation, we can associate a set of truth assignments, namely those that do not satisfy \( \psi_i \), i.e., \( M_i = \{ v \mid [v]_i = F \} \). The invariant we will maintain as we build the circuit line by line, is that \( C_i \) “correctly labels” assignments belonging to \( M_i \). That is, for \( v \in M_i \), if \( C_i(v) = F \) then \( v \) does not satisfy some clause \( A_i(\overline{P}, \overline{Q}) \) and if \( C_i(v) = T \) then \( v \) does not satisfy some clause \( B_j(\overline{Q}, \overline{P}) \). Notice that the last line \( \psi_n = \{ \} \), and so \( M_n \) is the set of all truth assignments. Thus, if the invariant is maintained, \( C_n \) will indeed be an interpolant because it will “correctly” label all assignments.

Let us now describe how we construct the circuit. For each line \( \psi_e \), we will add a line \( P_e = E_e \). What \( E_e \) is will depend on the justification for the line \( \psi_e \) in the refutation. Let us begin with the cases when \( \psi_e \) is a clause belonging to \( \Gamma \). There are two cases to consider.
• Suppose $\psi_e \in \{A_i(\overrightarrow{p}, \overrightarrow{q})\}_{i=1}^k$. Observe that any $v \in M_e$ does not satisfy a clause in $\{A_i(\overrightarrow{p}, \overrightarrow{q})\}_{i=1}^k$, and could be safely labeled $F$. Thus, the line corresponding to $\psi_e$ will be $P_e = F$.
• If $\psi_e \in \{B_j(\overrightarrow{q}, \overrightarrow{r})\}_{j=1}^f$, then each $v \in M_e$ could be labeled $T$ because it does not satisfy $\psi_e \in \{B_j(\overrightarrow{q}, \overrightarrow{r})\}_{j=1}^f$. Thus, we have $P_e = T$.

The next cases to consider are when $\psi_e$ is a resolvent of two clauses. So let $\psi_e = p_1 \cup p_2$ and let it be the resolvent of clauses $\psi_a$ and $\psi_b$ in the refutation. We need to consider different cases based on the proposition with respect to which $\psi_e$ is a resolvent. Let us begin by considering the case when the proposition being resolved is one of the common variables. Without loss of generality, let us take $\psi_a = p_1 \cup \{q\}$ and $\psi_b = p_2 \cup \{\neg q\}$. Consider an arbitrary assignment $v \in M_e$, i.e., $v[\psi_a] = F$. From the soundness of the resolution proof rule, we know that either $v \in M_a$ or $v \in M_b$. If $v(q) = F$ then $v \in M_a$; it may also, in addition, be the case that $v \in M_b$, but that is unimportant. We could label such assignments in the same manner as the labeling corresponding to line $\psi_a$, i.e., as per the value of variable $p_a$. Similarly, if $v(q) = T$ then $v \in M_b$ and so it can be labeled in the same manner as $p_b$. This gives us that the line corresponding to $\psi_e$ in this case should be $P_e = (\neg q \land p_a) \lor (q \land p_b)$.

Let us now consider the case when $\psi_e$ is a resolvent of $\psi_a$ and $\psi_b$, but the resolution step is taken with respect to a proposition (say $s$) that is either in $\overrightarrow{p}$ or in $\overrightarrow{r}$. Once again any $v \in M_e$ must also belong to $M_a \cup M_b$, but now since $s$ is not in $\overrightarrow{q}$, it cannot be explicitly mentioned in the line $P_e = E_v$, as we did in the previous case. Again, without loss of generality, let us assume that $\psi_e = p_1 \cup p_2$, $\psi_a = p_1 \cup \{s\}$, $\psi_b = p_2 \cup \{\neg s\}$. Let $v'$ be the assignment that is identical to $v$, except that it flips the assignment to $s$. Without loss of generality, we can assume that $v(s) = F$ and $v'(s) = T$, and for all other propositions $t$, $v(t) = v'(t)$. Observe that $v \in M_a$ and $v' \in M_b$. Further, $C_a(v) = C_a(v')$ and $C_b(v) = C_b(v')$; this is because $C_a$ and $C_b$ do not mention $s$. Let us consider two cases based on whether $s \in \overrightarrow{p}$ or $s \in \overrightarrow{r}$.

• Suppose $s \in \overrightarrow{p}$. Observe that in this case, for any $j$, $v[B_j] = v'[B_j]$, because $s$ does not appear in $B_j$. Now, since $C_a$ and $C_b$ satisfy our invariant, we have, if $C_a(v) = T(= C_a(v'))$ then for some $j$, $v[B_j] = F(= v'[B_j])$. Similarly, if $C_b(v') = T(= C_b(v))$ then for some $j$, $v[B_j] = F(= v'[B_j])$. Thus, if either $C_a$ or $C_b$ label $v$, $v'$ by $T$, then $C_e$ must do the same. Otherwise, both $C_a$ and $C_b$ label $v$ and $v'$ as $F$, and this common label is correct as per our invariant. Therefore, we have $P_e = P_a \lor P_b$ in this case.
• Now let us consider $s \in \overrightarrow{r}$. In this case, for any $i$, $v[A_i] = v'[A_i]$. Using an argument similar to the previous case, we can argue that if either $C_a$ or $C_b$ label $v$, $v'$ by $F$ then $C_e$ must do the same. Otherwise, $C_a$ and $C_b$ agree on the label, and that is indeed the correct label. Therefore, dually, in this case we have, $P_e = P_a \land P_b$.

The proof that the invariant is maintained follows inductively, from the arguments we have made for each case. Thus, $C_n$ is indeed the interpolant. It is worth noting that in $C_n$ we have a sequence of initial assignments $q_1 = ?, q_2 = ?, \ldots q_m = ?$ that assert that $\overrightarrow{q}$ are input variables. This seems to suggest that the size of $C_n$ also depends on
|\overrightarrow{q}|$ and hence on $\Gamma$. However, instead of asserting that all variables in $\overrightarrow{q}$ are input variables, we could assert only those variables in $\overrightarrow{q}$ that appear in the refutation. Thus, $|C_n|$ is indeed linear in the size of the refutation. \hfill \Box

**Example 2.26** Let us look at an example to see how an interpolant can be constructed from a refutation. Consider the set of clauses

\[
\Gamma = \{\{p, q\}, \{\neg p, r\}, \{\neg q, r\}, \{\neg r\}\}.
\]

Here $p$ is a proposition that only appears in the $A$-clauses, and $q$ and $r$ are propositions that appear in both $A$ and $B$ clauses. $\Gamma$ is unsatisfiable, and we are looking for an interpolant that only mentions the common variables $q, r$. Below we have the refutation (on the left) alongside the circuit for the interpolant (on the right) as per the proof of Theorem 2.25.

\[
\begin{align*}
q &= ? \\
\{\neg p, r\} & \quad P_1 = 0 \\
\{\neg r\} & \quad P_2 = 1 \\
\{\neg p\} & \quad P_3 = (\neg r \land P_1) \lor (r \land P_2) \\
\{\neg q, r\} & \quad P_4 = 1 \\
\{\neg q\} & \quad P_5 = (\neg r \land P_4) \lor (r \land P_2) \\
\{p, q\} & \quad P_6 = 0 \\
\{q\} & \quad P_7 = P_3 \lor P_6 \\
\{\}\ & \quad P_8 = (\neg q \land P_7) \lor (q \land P_5)
\end{align*}
\]

Observations like Theorem 2.25 have also been established for other proof systems of propositional logic. That is, in these proof systems, a short proof for a fact can be converted into a construction of a small interpolant. Theorem 2.25 can be strengthened for certain special sets of clauses — one can show that in certain special cases, not only is the interpolant small, but it is also monotonic.

**Definition 2.27 (Monotone Circuits)**

A monotone circuit $C$ is one where there are no assignments of the form $P_i = \neg P_j$.

A monotone circuit enjoys the following monotonic property. Let us say $v_1 \leq v_2$ if for all proposition $p$, if $v_1(p) = T$ then $v_2(p) = T$, i.e., $v_2$ sets at least as many propositions to $T$ as $v$. The value of a monotonic circuit with respect to this ordering on assignments is monotonic. In other words, if $C$ is monotone, then for any $v_1, v_2$ such that $v_1 \leq v_2$, we have $C(v_1) = T$ implies $C(v_2) = T$.

**Theorem 2.28** Let the collection of clauses $\Gamma = \{A_i(\overrightarrow{p}, \overrightarrow{q})\}_{i=1}^k \cup \{B_j(\overrightarrow{q}, \overrightarrow{q})\}_{j=1}^\ell$ have a resolution refutation of length $n$. Further assume that either $\overrightarrow{q}$ occur only positively in $A_i$s or $\overrightarrow{q}$ occur only negatively in $B_j$s. Then there is a monotone circuit $C(\overrightarrow{q})$ such that
\[ \bigwedge_i A_i(\overrightarrow{p}, \overrightarrow{q}) \models C(\overrightarrow{q}) \text{ and } C(\overrightarrow{q}) \land \bigwedge_j B_j(\overrightarrow{q}, \overrightarrow{r}) \text{ is unsatisfiable.} \]

In addition, \(|C|\) is \(O(n)\).

**Proof** The construction of the interpolant is identical to that in the proof of Theorem 2.25. All cases in that proof go through except for the case when the line \(\psi_e\) is a resolvent with respect to proposition \(q \in \overrightarrow{q}\), i.e., the common variables. In this case, in the proof of Theorem 2.25, the circuit was \(P_e = (\neg q \land P_a) \lor (q \land P_b)\), where \(\psi_e\) is the resolvent of lines \(\psi_a = \rho_1 \lor \{p\}\) and \(\psi_b = \rho_2 \cup \{\neg q\}\). This is not monotonic because if the use of \(\neg q\). To prove our result, we need to change the circuit in this case. We will change it by forcing it to be monotone in the most naïve way — we will remove the offending \(\neg q\) and write \(P_e = P_a \lor (q \land P_b)\).

The resulting construction is correct, but the inductive argument using the invariant from Theorem 2.25 does not go through! Let us see what the problem is. Recall, that for any line \(\psi_e\), we defined the set \(M_e = \{v \mid v[[\psi_e]] = F\}\). The invariant we proved in Theorem 2.25 was for any valuation \(v \in M_e\), we have

- If \(v[[C_e]] = F\) then \(v[[A_i]] = F\) for some \(i\), and
- If \(v[[C_e]] = T\) then \(v[[B_j]] = F\) for some \(j\).

Let us try to prove this invariant by induction as in the proof of Theorem 2.25. Consider the case that we just changed, i.e., of a resolvent with respect to a common variable. So \(\psi_e = \rho_1 \lor \rho_2\) is the resolvent of lines \(\psi_a = \rho_1 \lor \{q\}\) and \(\psi_b = \rho_2 \cup \{\neg q\}\). And we have, \(P_e = P_a \lor (q \land P_b)\). Consider a valuation \(v \in M_e\). The problem with carrying out this inductive proof occurs when \(v(q) = T\); the other case of \(v(q) = F\) goes through rather simply. Then \(v \in M_b\). Now if \(C_a(v) = F\) or \(C_b(v) = T\) then \(C_e(v) = C_b(v)\) and correctness follows from the inductive assumptions on \(C_b\). The problem occurs when \(C_a(v) = T\) and \(C_b(v) = F\).

The way to fix the problem is to prove a stronger invariant. In our old invariant, we proved that our circuit for line \(e\) was correct on the truth assignments in \(M_e = \{v \mid v[[\psi_e]] = F\}\). Our strengthening will show that the circuit for line \(e\) is correct on a larger set of truth assignments. Depending on whether we consider the case when \(\overrightarrow{q}\) appears only positively in \(\{A_i(\overrightarrow{p}, \overrightarrow{q})\}_i\), or the case when \(\overrightarrow{q}\) occurs only negatively in \(\{B_j(\overrightarrow{q}, \overrightarrow{r})\}_j\), the invariant (and the proof) is slightly different. We will present the proof only for the case when \(\overrightarrow{q}\) occurs negatively in \(\{B_j(\overrightarrow{q}, \overrightarrow{r})\}_j\). We will state the modified invariant for the other case, but leave the details to be filled out by the reader.

To describe the invariant in the case when \(\overrightarrow{q}\) occurs negatively in \(\{B_j(\overrightarrow{q}, \overrightarrow{r})\}_j\), we need to introduce some notation. For a clause \(\psi, \psi \vdash_{\overrightarrow{p}, \overrightarrow{q}}\) be the clause obtained by removing all literals involving propositions in \(\overrightarrow{q}\). On the other hand, \(\psi \vdash_{\overrightarrow{q}, \overrightarrow{r}}\) is the clause obtained from \(\psi\) by removing all literals of proposition \(\overrightarrow{p}\) as well as all positive literals of \(\overrightarrow{q}\). Our stronger invariant will be for every valuation \(v\)

- if \(v[[\psi \vdash_{\overrightarrow{p}, \overrightarrow{q}}]] = F\) and \(C_e(v) = F\) then \(v[[A_i]] = F\) for some \(i\), and
- if \(v[[\psi \vdash_{\overrightarrow{q}, \overrightarrow{r}}]] = F\) and \(C_e(v) = T\) then \(v[[B_j]] = F\) for some \(j\).

Notice that since \(\{\} \vdash_{\overrightarrow{p}, \overrightarrow{q}} = \{} \vdash_{\overrightarrow{q}, \overrightarrow{r}} = \{}\), proving this new invariant guarantees that \(C_n\) (where \(n\) is length of the resolution refutation) is an interpolant.
2.3 Craig’s Interpolation Theorem and Proof Complexity

We now argue that the stronger invariant holds for the new construction. We consider each case in order.

\( \psi_e \in \{A_i\}_i \): In this case, we have \( \psi_e \vdash \bar{\bar{\neg}} \neg \bar{\neg} \psi_e \) and \( P_e = F \). The invariant, therefore, holds.

\( \psi_e \in \{B_j\}_j \): Again we have \( \psi_e \vdash \bar{\neg} \neg \psi_e \). Since \( P_e = T \), the invariant holds.

**Resolvent w.r.t \( \bar{\neg} \bar{\neg} \)**: Let \( \psi_e = \rho_1 \cup \rho_2 \) be the resolvent of lines \( \psi_a = \rho_1 \cup \{q\} \) and \( \psi_b = \rho_2 \cup \{\neg q\} \). Recall we have \( P_e = P_a \lor (q \land P_b) \).

1. Consider \( v \) such that \( v[\psi_e \vdash \bar{\neg} \neg \psi_e] = F \) and \( C_e(v) = F \). In this case, if \( v(q) = T \) then \( v[\psi_b \vdash \bar{\neg} \neg \psi_b] = v[\psi_a \vdash \bar{\neg} \neg \psi_a] = F \). Also, since \( C_e(v) = F \), it must be that \( C_b(v) = F \), and so correctness follows by induction. On the other hand, if \( v(q) = F \) then \( v[\psi_a \vdash \bar{\neg} \neg \psi_a] = v[\psi_b \vdash \bar{\neg} \neg \psi_b] = F \). Also, \( C_a(v) = F \) and correctness follows by induction.

2. Consider \( v \) such that \( v[\psi_e \vdash \bar{\neg} \neg \psi_e] = F \) and \( C_e(v) = T \). If \( C_a(v) = T \) then since \( v[\psi_a \vdash \bar{\neg} \neg \psi_a] = v[\rho_1 \vdash \bar{\neg} \neg \psi_a] = F \), the invariant follows by induction. Notice, how the stronger invariant helped the proof go through in this case which was problematic before. On the other hand, if \( v[q \land C_b] = T \) then \( v(q) = T \). So \( v[\psi_b \vdash \bar{\neg} \neg \psi_b] = F \) and then invariant holds by induction.

**Resolvent w.r.t \( \bar{\neg} \)**: Let \( \psi_e = \rho_1 \cup \rho_2 \) be the resolvent of lines \( \psi_a = \rho_1 \cup \{p\} \) and \( \psi_b = \rho_2 \cup \{\neg p\} \). Recall \( P_e = P_a \land P_b \).

1. Suppose \( C_e(v) = F \) and \( v[\psi_e \vdash \bar{\neg} \psi_e] = F \). Then we know \( C_a(v) = C_b(v) = F \). Further either \( v[\psi_a] = F \) or \( v[\psi_b] = F \). Thus, correctness of construction by induction.

2. Suppose \( v[\psi_e \vdash \bar{\neg} \psi_e] = F \) and \( C_e(v) = T \). Now \( \psi_e \vdash \bar{\neg} \neg \psi_e = \psi_a \land \psi_b \). Let \( \psi_e \vdash \bar{\neg} \neg \psi_a \) and so \( v[\psi_a \vdash \bar{\neg} \neg \psi_a] = v[\psi_b \vdash \bar{\neg} \neg \psi_b] = F \). Further since \( C_e(v) = T \), either \( C_a(v) = T \) or \( C_b(v) = T \). So correctness follows by induction.

**Resolvent w.r.t \( \bar{\neg} \)**: Proof similar to previous case.

The proof of correctness when \( \bar{\neg} \bar{\neg} \) appears positively in \( \{A_i(\bar{\neg} \bar{\neg}, \bar{\neg} \bar{\neg})\}_i \) is similar, though the invariant is slightly different. For a clause \( \psi \), take \( \psi \vdash \bar{\neg} \neg \psi \) to be the clause obtained by removing literals of \( \bar{\neg} \bar{\neg} \) and negative literals of \( \bar{\neg} \bar{\neg} \). In addition, \( \psi \vdash \bar{\neg} \neg \psi \) is the clause obtained by removing literals of \( \bar{\neg} \bar{\neg} \). The invariant we will prove about the construction is, for every \( v \),

- if \( v[\psi \vdash \bar{\neg} \neg \psi] = F \) and \( C_e(v) = F \) then \( v[A_i] = F \) for some \( i \), and
- if \( v[\psi \vdash \bar{\neg} \neg \psi] = F \) and \( C_e(v) = T \) then \( v[B_j] = F \) for some \( j \).

The proof is similar and skipped. \( \square \)

**Example 2.29** Let us consider the set of clauses \( \Gamma \) from Example 2.26.

\[
\Gamma = \{\{p, q\}, \{\neg p, r\}, \{\neg q, r\}, \{\neg r\}\}.
\]
Notice that the common propositions, \( q \) and \( r \), appear only positively in \( A \). The refutation alongside the interpolant construction is as follows.

\[
q = \neg q, \\
\{ \neg p, r \} = P_1 = 0, \\
\{ \neg q \} = P_2 = 1, \\
\{ \neg p \} = P_3 = P_1 \lor (r \land P_2), \\
\{ \neg q, r \} = P_4 = 1, \\
\{ \neg q \} = P_5 = P_4 \lor (r \land P_2), \\
\{ p, q \} = P_6 = 0, \\
\{ q \} = P_7 = P_3 \lor P_6, \\
\{ \} = P_8 = P_7 \lor (q \land P_3)
\]

### 2.3.4 Lower bounds on Resolution Refutations

The results in Sect. 2.2 connecting resolution refutation lengths and size of interpolants, allows one to prove lower bounds on the length of resolution refutations for formulas. In particular we can show that there are sets of clauses \( \Gamma \) for which the shortest resolution refutations are exponential in the size of \( \Gamma \). Thus, not every unsatisfiable formula has a short proof in resolution. The specific example we consider relates to cliques in graphs and their coloring. Let us recall these classical problems.

Recall that in Proposition 1.35, we showed that determining if a graph is \( k \)-colorable can be reduced to checking the satisfiability of a set of formulas. More specifically, let us fix the graph \( G = (V, E) \) to have \( n \) vertices. Any such graph can be represented by an assignment to propositions \( \{q_{uv} \mid u, v \in \{1, 2, \ldots, n\}\} \), with the interpretation that \( (u, v) \in E \) iff \( q_{uv} \) is set to \( T \). Using \( r_{ui} \) to denote “vertex \( u \) has color \( i \)”, there is a set of clauses \( \text{color}_{n,k}(\vec{q}, \vec{r}) \) such that \( v \models \text{color}_{n,k}(\vec{q}, \vec{r}) \) iff the graph (over \( n \) vertices) represented by \( v \models \vec{q} \) has a \( k \)-coloring given by \( v \models \vec{r} \). The set \( \text{color}_{n,k} \) is almost identical to the construction given in the proof of Proposition 1.35. The only difference is that we will use the clause \( \neg q_{uv} \lor \neg r_{ui} \lor \neg r_{vj} \), for every pair of vertices \( u, v \in \{1, 2, \ldots, n\} \) and color \( i \in \{1, 2, \ldots, k\} \), instead of \( \neg r_{ui} \lor \neg r_{vj} \) for every edge \( (u, v) \) as given in Proposition 1.35. Note that \( \text{color}_{n,k} \) has \( O(n^2k + nk^2) \) clauses.

Whether a graph is \( k \)-colorable is related to the presence of graph structures called **cliques**.

**Definition 2.30** A \( k \)-clique in a graph \( G = (V, E) \) is a subset \( U \subseteq V \) such that \( |U| = k \) and for every \( u, v \in U \), with \( u \neq v \), we have \( (u, v) \in E \).

Like graph coloring, the problem of determining if a graph has a \( k \)-clique can be reduced to satisfiability.
Proposition 2.31 For any $n, k$, there is a set of $O(n^2k^2)$ clauses $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ such that $\nu \models \text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ iff the graph represented by $\nu \vdash \overrightarrow{q}$ has a $k$-clique given by $\nu \vdash \overrightarrow{p}$

Proof The proof of this observation is similar to that of Proposition 1.35. We will introduce propositions that encode the $k$-clique, and clauses will specify constraints that characterize properties of a $k$-clique. Let proposition $p_{iu}$, for $i \in \{1, \ldots, k\}$ and $u \in \{1, \ldots, n\}$, denote that “the $i$th vertex in clique is $u$”. Then $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q})$ is the following set of clauses:

- For each $1 \leq i \leq k$, the clause $p_{1i} \lor p_{2i} \lor \cdots \lor p_{in}$. These clauses capture the constraint that the $i$th vertex of the clique must be among $\{1, \ldots, n\}$.
- For each $1 \leq i \leq k$, and $1 \leq u, v \leq n$ such that $u \neq v$, the clause $\lnot p_{iu} \lor \lnot p_{iv}$.

Intuitively, this says that the $i$th vertex of the clique can be at most one vertex.

- For each $1 \leq i, j \leq k$ and $1 \leq u, v \leq n$ with $u \neq v$, we have the clause $\lnot p_{iu} \lor \lnot p_{jv} \lor q_{uv}$. These clauses together say that if $u, v$ are vertices in the clique then they have an edge between them.

The proof that these clauses satisfy the proposition is left to the reader. □

Observe that if a graph $G$ has a $k$-clique then it cannot be colored using $k - 1$ colors. This is because each of the vertices in the clique must get different colors. Thus a graph with a $k$-clique needs at least $k$ colors. This leads us to the following observation.

Proposition 2.32 For any $n, k$, $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ is unsatisfiable.

Proof A satisfying assignment for $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ is a graph encoded by $\overrightarrow{q}$ that has $k$-clique (identified by $\overrightarrow{p}$) and can be colored using $k - 1$ colors (with the coloring encoded by $\overrightarrow{r}$). This is clearly impossible. □

Since $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ is unsatisfiable, it must have a resolution refutation. How long is its refutation? Our goal will be to prove that this is exponential in $n$. Since the size of $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ itself is polynomial in $n$, this would be an example that has a “long” proof. In order to establish this result, we present a celebrated result in circuit complexity whose proof can be found in textbooks like [?].

Theorem 2.33 (Razborov, Alon-Bopanna)

Any monotone circuit that evaluates to $\top$ on $n$-vertex graphs containing a $k$-clique and evaluates to $\bot$ on $n$-vertex graphs that are $k - 1$ colorable must have size at least $2^{\Omega(\sqrt{n})}$, when $k \leq n^3$.

Theorem 2.33 combined with Theorem 2.28 gives us the desired lower bound on proof lengths.

Theorem 2.34 Any resolution refutation of $\text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{r})$ must have length at least $2^{\Omega(\sqrt{n})}$, when $k \leq n^3$. 
Proof Let there be a resolution refutation of length \( \ell \). Observe that \( \overrightarrow{q} \) appears only positively in \( \text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \) and only negatively in \( \text{color}_{n,k}(\overrightarrow{q}, \overrightarrow{p}) \). Thus, \( \text{clique}_{n,k}(\overrightarrow{p}, \overrightarrow{q}) \cup \text{color}_{n,k-1}(\overrightarrow{q}, \overrightarrow{p}) \) satisfies the conditions of Theorem 2.28, and so there is a monotone interpolant of size \( O(\ell) \). The interpolant is a monotone circuit satisfying conditions of Theorem 2.33. Thus, \( \ell \) must be at least \( 2^{\Omega(\sqrt{\ell})} \). \( \square \)

Theorem 2.34 establishes that resolution proofs can be long as an exponential function of the size of the input clauses. Historically, the above theorem was not the first example of a proof that some formulas have long resolution proofs. The first such result was established by Haken. He showed that the pigeon hole principle has long resolution proofs. Recall that the pigeon hole principle says that if there are \( n \) holes and \( n + 1 \) pigeons then some hole may contain more than one pigeon. Let \( \text{pigeonhole}_n \) be the propositional logic formula that says that every pigeon goes to a hole and no hole contains more than one pigeon. Then \( \text{pigeonhole}_n \) is unsatisfiable, and Haken showed that any resolution refutation of this fact must be exponential in \( n \). The broad principle of using interpolation and lower bounds from monotonic circuit complexity have been used to establish that other proof systems can also have long proofs.

Understanding how long resolution proofs can be, and when they can be long, helps our theoretical understanding of the limits of SAT solvers — what examples they may work well and when they can take long. But a probably more important reason for studying proof lengths in some proof system is because of its connections to some fundamental question in complexity theory. Essentially, the goal in proof complexity is to understand if there is a proof system for propositional logic that has the property that all facts have short proofs. Investigating whether this is true or not has important implications in complexity theory. Let us see why.

Definition 2.35 A proof system \( \Pi \) is super if every tautology \( \varphi \) has a proof in \( \Pi \) such that length of the proof is bounded by a polynomial function of \( |\varphi| \).

Now Theorem 2.34 says that resolution is not a super proof system. But are there other proof systems that are super? This is intimately tied to another open question in complexity theory.

Theorem 2.36 (Cook-Reckhow)
Propositional logic has a super proof systems if and only if \( \text{NP} = \text{coNP} \).

Proof There are two directions to this proof. Assume that there is a super proof system. Then the problem to determine if a given formula is valid, is in \( \text{NP} \) — the \( \text{NP} \) algorithm simply guesses the proof and checks that it is a proof in our super proof system. Since the problem of checking validity is \( \text{coNP} \)-complete, it follows that \( \text{coNP} = \text{NP} \); the \( \text{NP} \) algorithm for an arbitrary problem \( \Lambda \in \text{coNP} \) is simple to reduce it to validity checking and then use the \( \text{NP} \) algorithm based on the super proof system.

On the other hand, suppose \( \text{NP} = \text{coNP} \) then there is nondeterministic Turing machine \( N \), running in polynomial time, that checks if a given propositional logic formula is valid. Proofs for a formula \( \varphi \) in our new “super” proof system will simply
be the nondeterministic choices that cause $N$ to accept $\varphi$; notice, that these proofs will be polynomially long because $N$ only has computations that are polynomially long.

In the light of Theorem 2.36, to resolve the NP versus coNP question, we need to prove that there are no super proof systems for proposition logic. Cook proposed an approach to tackling this problem. Consider concrete natural proof systems for propositional logic, one by one, and show that they are not super. Then use the intuition developed in this process to generalize and prove the absence of any super proof system. Resolution was the first proof system shown to be not super. Since then other proof systems have also been proved to be not super. However, there are still natural proof systems for which we have not been able to prove exponential lower bounds for proof lengths. One such proof system is the Frege proof system we introduced. It is still open whether there are tautologies for which proofs in the Frege proof system will be exponentially long.
Chapter 3
First Order Logic

Syntax, Semantics, and Overview

First order logic is a formal language to describe and reason about \textit{predicates}. Modern efforts to study this logic grew out of a desire to study the foundations of mathematics in number theory and set theory. It has a careful treatment of functions, variables, and quantification. First order logic deals with predicates as opposed to propositions — which are declarative statements that are either true or false — which is the subject of study in propositional logic. A predicate is a proposition that depends on the value of some variables. For example truth of the statement “\( x \) is prime”, depends on the value \( x \) takes (and of course also on the meaning of “is prime”). If \( x = 2 \) the statement “\( x \) is prime” would be true and if \( x = 4 \) it would be false. Predicates may depend on more than one variable. For example, the truth of \( P(x, y) \triangleq x + y = 0 \) depends on the values of both \( x \) and \( y \).

One way to convert a predicate into a proposition by substituting values for the predicate variables. For example, for the predicate \( P \) defined in the previous paragraph, \( P(2, -2) \) denotes the proposition “\( 2 + (-2) = 0 \)”. Another way to obtain propositions in predicate logic is by using \textit{quantifiers}, which allows one to express statements like the predicate holds for all values of the variable, or the predicate holds for some values of the variable. In this chapter, we introduce the syntax and semantics of first order logic, and some of the questions we will explore in this book.

3.1 Syntax

First order logic formulas are defined over a \textit{vocabulary} or \textit{signature} that identifies the predicates and constants that can be used in the formulas.

\textbf{Definition 3.1} A \textit{vocabulary} or \textit{signature} is \( \tau = (C, \mathcal{R}) \), where \( C = \{c_1, c_2, \ldots \} \) is a set of \textit{constant} symbols, and \( \mathcal{R} = \{\mathcal{R}^k\}_k \) is a collection of sets with \( \mathcal{R}^k = \{R_1^k, R_2^k, \ldots \} \) is a set of \( k \)-ary relation symbols.

Note that any of the above sets of constants, or \( k \)-ary relation symbols can be empty, finite, or infinite. A signature will be called \textit{relational} if there are no constant
symbols in the signature. A signature is finite if the total number of symbols in the signature is finite.

We will typically consider signatures that are finite. When the arity of a relation symbol is clear from the context, we will drop the superscript.

Formulas in first-order logic over signature $\tau$ are sequences of symbols, where each symbol is one of the following.

1. The symbol $=$
2. An element of the infinite set $V = \{x_1, x_2, x_3, \ldots\}$ of variables
3. Constant symbols and relation symbols in $\tau$
4. The symbol $\neg$ called negation
5. The symbol $\lor$ called disjunction
6. The symbol $\exists$ called the existential quantifier
7. The symbols $($ and $)$ called parenthesis

As always, not all such sequences are formulas; only well formed sequences are formulas in the logic. This is defined as follows.

**Definition 3.2** A well formed formula (wff) over signature $\tau$ is inductively defined as follows.

1. If $t_1, t_2$ are either variables or constant symbols in $\tau$ then $t_1 = t_2$ is a wff.
2. If $t_i$ is either a variable or a constant for $1 \leq i \leq k$ and $R$ is a $k$-ary relation symbol in $\tau$ then $Rt_1t_2 \cdots t_k$ is a wff.
3. If $\phi$ is a wff then $(\neg \phi)$ is a wff.
4. If $\phi$ and $\psi$ are wffs then $(\phi \lor \psi)$ is a wff.
5. If $\phi$ is a wff and $x$ is a variable then $(\exists x \phi)$ is a wff.

More succinctly, we could capture the above definition by the following BNF grammar.

$$\phi ::= t = t \mid Rtt \cdots t \mid (\neg \phi) \mid (\phi \lor \psi) \mid (\exists x \phi)$$

where $t$ is either a constant symbol or a variable, and $R$ is a relation symbol, and $x$ is a variable.

It is useful to introduce logical operators in addition to those in Definition 3.2. These operators can be “syntactically” defined in terms of the operators in Definition 3.2. As in propositional logic, we can define the Boolean connectives conjunction as $\phi \land \psi = (\neg (\neg \phi) \lor (\neg \psi))$, implication as $\phi \rightarrow \psi = ((\neg \phi) \lor \psi)$, true as $\top = (\phi \lor (\neg \phi))$, and false as $\bot = (\neg \top)$. Finally, we can define universal quantification as $(\forall x \phi) = (\neg (\exists x (\neg \phi)))$.

To avoid too many parenthesis, and at the same time have an unambiguous interpretation of formulas, we will assume the following precedence of operators (from increasing to decreasing): $\neg, \land, \lor, \rightarrow, \forall, \exists$. Thus $\forall x \forall y x = y \rightarrow \neg Rxy$ means $(\forall x (\forall y (x = y \rightarrow (\neg Rxy))))$. We will also drop the outermost parentheses, and since $\land$ and $\lor$ are associative, drop parentheses in formulas involving the conjunction/disjunction of multiple formulas.
Example 3.3 Consider signature $\tau = \{R\}$ where $R$ is a binary relation symbol. The following are formulas over this signature:

- Reflexivity: $\forall x Rxx$
- Irreflexivity: $\forall x (\neg Rxx)$
- Symmetry: $\forall x \forall y (Rxy \rightarrow Ryx)$
- Anti-symmetry: $\forall x \forall y ((Rxy \land Ryx) \rightarrow x = y)$
- Transitivity: $\forall x \forall y \forall z ((Rxy \land Ryz) \rightarrow Rxz)$

Non-examples of formulas include $Rx$ ($R$ expects two arguments); $x$ (a variable is not a formula); $(Rxy \lor Rzx$ (mismatched parentheses); $\exists x$ (x is quantified but there is no formula provided as argument).

3.2 Semantics

The semantics of formulas in any logic is defined with respect to a model. In the context of propositional logic, models were truth assignments to the propositions. For first order logic, models will be objects that help identify the interpretation of constants and relation symbols. Such models are called structures.

Definition 3.4 A structure $\mathcal{A}$ of signature $\tau$ is $\mathcal{A} = (A, \{c^\mathcal{A}\}_{c \in \tau}, \{R^\mathcal{A}\}_{R \in \tau})$ where

- $A$ is a non-empty set called the domain/universe of the structure,
- For each constant symbol $c \in \tau$, $c^\mathcal{A} \in A$ is its interpretation,
- For each $k$-ary relation symbol $R \in \tau$, $R^\mathcal{A} \in A^k$ is its interpretation.

The structure $\mathcal{A}$ is said to be finite if the universe $A$ is finite. The universe of a structure $\mathcal{A}$ will be denoted by $u(\mathcal{A})$.

Many mathematical objects can be studied through the lens of logic. Let us look at some example signatures and structures.

Example 3.5 Consider the signature $\tau_G = \{E\}$, where $E$ is a binary relation. We use this signature to study graphs. A graph $H = (V, E)$ modeled as a structure is $G = (G, E^\mathcal{A})$, where the universe $G$ is the set of vertices $V$, and for a pair of vertices $u, v \in G$ ($= V$), $E^\mathcal{A}uv$ holds iff $(u, v) \in E$.

Example 3.6 Let $\tau_O = \{<, S\}$ where $<$ and $S$ are binary relation symbols. A finite order structure is $O = (O, <^O, S^O)$, where $O$ is the universe of elements, $<$ is interpreted to be an ordering relation, and $S$ as the “successor” relation.

Example 3.7 Labeled binary trees, where vertices are labeled by elements of $\Sigma$, can be represented as a structure in the following manner. Let $\tau_T = \{<, S_0, S_1, (Q_a)_{a \in \Sigma}\}$ where $<, S_0, S_1$ are binary relation symbols, $Q_a$ is a unary relation symbol. A tree (labeled by symbols in $\Sigma$) is a structure $T = (T, <^T, (S_0^T, S_1^T, (Q_a^T)_{a \in \Sigma})$ where
elements of $T$ are called vertices, $<$ is the ancestor relation, $S_0$ and $S_1$ are the left and right child relations, respectively, and $Q_0$ holds in all vertices labeled by $a$.

For example, consider the binary tree shown in Fig. 3.1. Let us see how this tree is represented as a structure. The universe will consist of the vertices of the tree. We could use any names for the vertices. But it is convenient to name them in a manner that makes the edge relation explicit. The root will be $Y$, and for a vertex $w$, its left child will be $w0$, while its right child will be $w1$. Given this, the tree in Fig. 3.1 corresponds to the following structure:

$$T = \{Y, P, Q, PP, PQ, QP, QQ\},$$

$$\preceq_T = \{(u, u0) | u \in \{Y, 0, Q\}\},$$

$$S_0^T = \{(u, u0) | u \in \{Y, 0, 1\}\}, Q_0 = \{0, 01, 11\}, Q_1 = \{0, 01, 10\}.$$  

In order to define the semantics of a first order logic formula, we need a structure, and an assignment. An assignment maps every variable to an element in the universe of the structure.

**Definition 3.8** For a $\tau$-structure $\mathcal{A}$, an assignment over $\mathcal{A}$ is a function $\alpha : V \rightarrow u(\mathcal{A})$ that assigns every variable $x \in V$ a value $\alpha(x) \in u(\mathcal{A})$. If $t$ is a constant symbol $c$, we will take $\alpha(t)$ to be $c^\mathcal{A}$.

For an assignment $\alpha$ over $\mathcal{A}$, $\alpha[x \mapsto a]$ is the assignment

$$\alpha[x \mapsto a](y) = \begin{cases} 
\alpha(y) & \text{for } y \neq x \\
 a & \text{for } x = y
\end{cases}$$

We now have all the elements to define the semantics of a formula. The satisfaction relation will be a ternary relation — $\mathcal{A} \models \varphi[\alpha]$ to be read as “$\varphi$ is true/holds in $\mathcal{A}$ under assignment $\alpha$”. The relation will be defined inductively on the structure of the formula. In defining the relation, we will also say $\mathcal{A} \not\models \varphi[\alpha]$ to mean that $\mathcal{A} \models \varphi[\alpha]$ does not hold.

**Definition 3.9** The relation $\mathcal{A} \models \varphi[\alpha]$ is inductively defined as follows.

* $\mathcal{A} \models t_1 = t_2[\alpha]$ iff $\alpha(t_1) = \alpha(t_2)$
* $\mathcal{A} \models R t_1 \cdots t_n[\alpha]$ iff $(\alpha(t_1), \alpha(t_2), \ldots, \alpha(t_n)) \in R^\mathcal{A}$
* $\mathcal{A} \models (\neg \varphi)[\alpha]$ iff $\mathcal{A} \not\models \varphi[\alpha]$ 
* $\mathcal{A} \models (\varphi \lor \psi)[\alpha]$ iff $\mathcal{A} \models \varphi[\alpha]$ or $\mathcal{A} \models \psi[\alpha]$ 
* $\mathcal{A} \models (\exists x \varphi)[\alpha]$ iff for some $a \in u(\mathcal{A})$, $\mathcal{A} \models \varphi[a[x \mapsto a]]$ 

1 For a relation symbol $R$, we will sometimes write $R^\mathcal{A}a_1a_2 \cdots a_n$ instead of $(a_1, a_2, \ldots, a_n) \in R^\mathcal{A}$.
3.2 Semantics

Example 3.10 Consider a structure (graph) over the vocabulary of graphs \( \tau_G = \{E\} \) \( \mathcal{G} = \{(1, 2, 3, 4), E^G = \{(1, 2), (2, 3), (3, 4), (4, 1)\} \). For any assignment \( \alpha \), \( \mathcal{G} \models \forall x \exists y \text{Exy}[\alpha] \) because

\[
\begin{align*}
\mathcal{G} &\models \exists y \text{Exy}[\alpha[x \mapsto 1]] \quad \text{because} \\
\mathcal{G} &\models \text{Exy}[\alpha[x \mapsto 1][y \mapsto 2]], \\
\mathcal{G} &\models \exists y \text{Exy}[\alpha[x \mapsto 2]] \quad \text{because} \\
\mathcal{G} &\not\models \text{Exy}[\alpha[x \mapsto 2][y \mapsto 3]], \\
\mathcal{G} &\models \exists y \text{Exy}[\alpha[x \mapsto 3]] \quad \text{because} \\
\mathcal{G} &\not\models \text{Exy}[\alpha[x \mapsto 3][y \mapsto 4]], \quad \text{and} \\
\mathcal{G} &\models \exists y \text{Exy}[\alpha[x \mapsto 4]] \quad \text{because} \\
\mathcal{G} &\not\models \text{Exy}[\alpha[x \mapsto 4][y \mapsto 1]].
\end{align*}
\]

Notice that in Example 3.10, the actual assignment to variables \( x \) and \( y \) did not matter when determining the satisfaction of the formula in the graph. This is because they are *bound* by the universal and existential quantifiers in the formula \( \varphi \). This leads us to the important notion of bound and free variables in a formula. We begin by defining the scope of a quantifier.

Definition 3.11 For a wff \( \varphi = (\exists x \psi) \), \( \psi \) is said to be the *scope* of the quantifier \( \forall x \).

Definition 3.12 Every occurrence of the variable \( x \) in \( \varphi = (\exists x \psi) \) is called a *bound occurrence* of \( x \) in \( \varphi \).

Any occurrence of \( x \) which is not bound is called a *free occurrence* of \( x \) in \( \varphi \).

The free variables in wff \( \varphi \) will be denoted by \( \text{free}(\varphi) \). The notation \( \varphi(x_1, x_2, \ldots, x_n) \) will be used to indicate that \( \text{free}(\varphi) \subseteq \{x_1, \ldots, x_n\} \).

Let us look at an example to understand the subtle definition of bound and free variables.

Example 3.13 Consider \( \varphi = Pxy \lor (\exists x (\exists y Rx y) \lor Qxy) \) the free variables are shown in **bold**. Notice that a variable may occur both bound and free. As we will establish soon, we can change the names of bound variables without affecting the meaning of formulas. Thus \( \psi = Pxy \lor (\exists u (\exists v Ruv) \lor Quv) \) is an equivalent formula. Therefore, we will typically assume that bound and free variables are disjoint. In addition, since bound variables can be renamed without affecting its meaning, we can also assume that every bound variable is in the scope of a unique quantifier. Thus, instead of \( Pxy \lor (\exists u (\exists v Ruv) \lor (\exists u Quv)) \), we will consider the equivalent formula \( Pxy \lor (\exists u (\exists v Ruv) \lor (\exists u Quv)) \)

The satisfaction of a formula in a structure \( \mathcal{A} \) under assignment \( \alpha \) only depends on the values \( \alpha \) assigns to the free variables; the values assigned to the bound variables in \( \alpha \) are unimportant.

Theorem 3.14 For a formula \( \varphi \) and assignments \( \alpha_1 \) and \( \alpha_2 \) such that for every \( x \in \text{free}(\varphi) \), \( \alpha_1(x) = \alpha_2(x) \), \( \mathcal{A} \models \varphi[\alpha_1] \iff \mathcal{A} \models \varphi[\alpha_2] \).
Theorem 3.14 can be proved by induction on the structure of the formula $\varphi$. The proof is left as an exercise for the reader. Theorem 3.14 suggests that if a formula has no free variables, its truth is independent of the assignment. Formulas without any free variables (i.e., those all of whose variables are bound) are an important class of formulas and have special name.

**Definition 3.15** A sentence is a formula $\varphi$ none of whose variables are free, i.e., $\text{free}(\varphi) = \emptyset$.

An immediate consequence of Theorem 3.14 is that the truth of sentences is independent of the assignment.

**Proposition 3.16** For a sentence $\varphi$, and any two assignments $\alpha_1$ and $\alpha_2$, $\mathcal{A} \models [\alpha_1] \varphi$ iff $\mathcal{A} \models [\alpha_2] \varphi$.

Proposition 3.16 is an immediate consequence of Theorem 3.14. Thus, we say $\mathcal{A} \models \varphi$ whenever $\mathcal{A} \models [\alpha] \varphi$ for some $\alpha$.

**Definition 3.17** For a sentence $\varphi$, $\mathcal{A}$ is said to be a model of $\varphi$ iff $\mathcal{A} \models \varphi$. We will denote by $[\varphi]$ the set of all models of $\varphi$.

Satisfiability and validity/tautologies are defined in a manner similar to that for propositional logic — a formula is satisfiable if there is some model and assignment in which it is true, and it is valid if it is true in all models and assignments.

**Definition 3.18** A formula $\varphi$ over signature $\tau$ is said to be satisfiable iff for some $\tau$-structure $\mathcal{A}$ and assignment $\alpha$, $\mathcal{A} \models [\alpha] \varphi$.

A formula $\varphi$ over signature $\tau$ is said to be logically valid iff for every $\tau$-structure $\mathcal{A}$ and assignment $\alpha$, $\mathcal{A} \models [\alpha] \varphi$. We will denote this by $\models \varphi$.

We can also define with a formula $\varphi$ is a logical consequence of a set of formulas $\Gamma$ in exactly the same way as we defined it for propositional logic.

**Definition 3.19** For a set of formulas $\Gamma$, we say $\mathcal{A} \models [\alpha] \Gamma$ iff for every $\varphi \in \Gamma$, $\mathcal{A} \models [\alpha] \varphi$.

We say $\varphi$ is a logical consequence of $\Gamma$, denoted by $\Gamma \models \varphi$, if and only if for every $\mathcal{A}$ and $\alpha$, $\mathcal{A} \models [\alpha] \Gamma$ implies that $\mathcal{A} \models [\alpha] \varphi$. Thus, if $\emptyset \models \varphi$ then $\models \varphi$.

The following observation is an immediate consequence of the definition of logical consequence.

**Proposition 3.20** $\Gamma \cup \{\varphi\} \models \psi$ iff $\Gamma \models \varphi \rightarrow \psi$

Finally two formulas are (semantically) equivalent, if the hold in exactly the same set of structures and assignments.

**Definition 3.21** Formulas $\varphi$ and $\psi$ are said to be logically equivalent (denoted $\varphi \equiv \psi$) if for every $\mathcal{A}$ and assignment $\alpha$, $\mathcal{A} \models [\alpha] \varphi$ iff $\mathcal{A} \models [\alpha] \psi$. 
Appendix A
Computability and Complexity Theory
A Brief Primer

We will now review the basic definitions and theorems in the area of computational complexity, which tries to study various models of computation with the goal of understanding their relative computational power, and classify computational problems in terms of computational resources they need. Here, we will primarily consider time and space as the principal resources we will measure for an algorithm.

Recall that the computational problems one studies in the context of theoretical computer science are usually decision problems. Decision problems are those where given an input, one expects a Boolean answer. Typically, input instances are encoded as strings over some alphabet of symbols. A decision problem partitions inputs into those for which the expected answer is “yes”/”true” and those for which the answer is “no”/”false”. Therefore, a decision problem is often identified with a language, or a collection of strings, namely, those for which the problem demands a “yes” answer. Similarly, the machines we will define, will answer “yes”/”accept” or “no”/”reject” on input strings, and we associate a language \( L(M) \) with machine \( M \), which is the collection of all strings it accepts. Given this interpretation of problems and machines, we will typically say that a machine \( M \) solves a problem \( L \) (or rather accepts/recognizes) if \( L = L(M) \), i.e., \( M \) answers “yes” on exactly the inputs that the problem demands the answer to be “yes”.

The main model of computation that we will consider is that of a Turing machine. However before introducing this model, let us recall some of the notation on strings and languages that we will use.

Alphabet, Strings, and Languages.

An alphabet \( \Sigma \) is a finite set of elements. A (finite) string over \( \Sigma \) is a (finite) sequence \( w = a_0a_1 \cdots a_k \) over \( \Sigma \) (i.e., \( a_i \in \Sigma \), for all \( i \)). The length of a string \( w = a_0a_1 \cdots a_k \), denoted \( |w| \), is the number of elements in it, which in this case is \( k + 1 \). The unique string of length 0, called the empty string, will be denoted by \( \epsilon \). For a string \( w = a_0a_1 \cdots a_k \), the \( i \)th symbol of the string \( a_i \) will be denoted...
as $w[i]$. For strings $u = a_0a_1\cdots a_k$ and $v = b_0b_1\cdots b_m$, their concatenation is the string $uv = a_0a_1\cdots a_kb_0b_1\cdots b_m$. The set of all (finite) strings over $\Sigma$ is denoted by $\Sigma^*$; we will sometimes use $\Sigma^i$ to denote the set of strings of length $i$. A language $A$ is a set of strings, i.e., $A \subseteq \Sigma^*$. Given languages $A, B$, their concatenation $AB = \{uv \mid u \in A, v \in B\}$. For a language $A$, $A^0 = \{\varepsilon\}$, and $A^i$ denotes the $i$-fold concatenation of $A$ with itself, i.e., $A^i = \{u_1u_2\cdots u_i \mid \forall j. u_j \in A\}$. Finally, the Kleene closure of a language $A$, is $A^* = \bigcup_{i \geq 0} A^i$.

### A.1 Turing Machines

We now recall the definition of a Turing machine. Since we will use this model to define the time and space bounds during a computation, as well as define computable functions, the most convenient model to consider is that of a multi-tape Turing machine shown in Fig. A.1. Such a model has a read-only input tape, a write-only output tape and finitely many read/write work tape, and a write-only output tape. Intuitively, the machine works as follows. Initially, the input string is written out on the input tape, and all the remaining tapes are blank. The tape heads are scanning the leftmost cell of each tape, which we will refer to as cell $0$. This cell contains a special symbol $\triangleright$ in every take except the output tape. This is the left end marker, which helps the machine realize which cell is the leftmost cell. We will assume these cells are never overwritten by any other symbol, and whenever $\triangleright$ is read on a particular tape, the tape head of the Turing machine will move right. At any given step of the Turing machine does the following. Based on the current state of its finite control, and symbols scanned by each tape head, the machine will change the state of its finite control, write new symbols on each of the work tapes, and move it’s heads on the input and work tapes either one cell to the left or one cell to the right. During the step, the machine may also choose to write some symbol on its output tape. If

---

1 Here we are assuming the the 0th symbol is the “first”.

Fig. A.1 Turing machine with a read-only input tape, finitely many read/write worktapes, and a write-only output tape.
it writes something on the output tape, then the output tape head moves one cell to the right. If it does not write anything, then the output tape head does not move. We will assume that the machine has two special halting states — \( q_{\text{acc}} \) and \( q_{\text{rej}} \) — with the property that the machine cannot take any further steps from these states. These are captured in the formal definition of deterministic Turing machines below.

**Definition A.1** A deterministic Turing machine with \( k \) work tapes is a tuple \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \sqcup, \triangleright) \) where

- \( Q \) is a finite set of control states
- \( \Sigma \) is a finite set of input symbols
- \( \Gamma \supseteq \Sigma \) is a finite set of tape symbols. We assume that \( \{\sqcup, \triangleright\} \subseteq \Gamma \setminus \Sigma \).
- \( q_0 \in Q \) is the initial state
- \( q_{\text{acc}} \in Q \) is the accept state
- \( q_{\text{rej}} \in Q \) is the reject state, with \( q_{\text{rej}} \neq q_{\text{acc}} \) and
- \( \delta : (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma^{k+1} \to Q \times \{-1, +1\} \times (\Gamma \times \{-1, +1\})^k \times (\Gamma \cup \{\varepsilon\}) \) is the transition function; here \(-1\) indicates moving the head one position to the left and \(+1\) indicates moving the head one position to the right. For \( \delta(q, \gamma_0, \gamma_1, \ldots, \gamma_k) = (q, d_0, \gamma'_1, d_1, \gamma'_2, d_2, \ldots, \gamma'_k, d_k, o) \), for any \( i \in \{0, 1, \ldots, k\} \), if \( \gamma_i = \triangleright \) then \( \gamma'_i = \triangleright \) and \( d_i = +1 \).

We will now formally describe how the Turing machine computes. For this we begin by first identifying information about the Turing machine that is necessary to determine its future evolution. This is captured by the notion of a configuration. A single step of a Turing machine depends on all the factors that determine which transition is taken. This clearly includes the control state, and the symbols being read on the input tape and the work tape. However this is not enough. The contents of the work tape change, and what is stored influences what will be read in a future step. Thus we need to know what is stored in each cell of the work tape. Since the input tape is read-only, its contents remain static and so we don't need to carry around its contents. We also need to know the position of each tape head because that determines what is read in this step, how the contents of a tape will change based on the current step, and what will be read in the future as the heads move. Because of all of these observations, a configuration of a Turing machine is taken to be the control state, the position of the input head, the contents of the work tape, and the position of the work tape head. The work tape contents and head position is often represented as a single string where a special marker indicates the head position. These are captured formally by the definition below.

**Definition A.2 (Configurations)**

A configuration \( c \) of a Turing machine \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \sqcup, \triangleright) \) is a member of the set \( Q \times \Sigma \times \Gamma^* \times \Gamma^{\sqcup \cdot \sqcup^k} \times \Gamma^{\triangleright \cdot \sqcup^k} \), where we assume that \( * \notin \Gamma \) indicates the position of the head. For example, a configuration \( c = (q, i, u_1 a_1 v_1 \sqcup \sqcup^a, u_2 a_2 v_2 \sqcup \sqcup^a) \)

\( \sqcup^a \) is an infinite sequence of blank symbols. Recall that almost all cells contain \( \sqcup \) and so the tape contents are a string of the form \( u \sqcup \sqcup^a \), where \( u \) is initial portion of the tape containing some non-blank symbols.
is the configuration of a 2-work tape Turing machine, whose control state is currently
$q$, the input head is scanning cell $i$, work tape $i$ ($i \in \{1, 2\}$) contains $u_i$ to left of
the head, head is scanning symbol $a_i$ and $v_i \in \Gamma^{\omega}$ are the contents of cells to the right of
the head.

The initial configuration (the configuration of the Turing machine when it starts)
is $(q_0, 0, * \mathrel{\triangleleft} \sqcup^{\omega}, \ldots, * \mathrel{\triangleleft} \sqcup^{\omega})$. An accepting configuration is a member of the set
$q_{\text{acc}} \times \mathbb{N} \times (\Gamma^{\ast} \{ * \} \Gamma^{\ast} \sqcup^{\omega})^{k}$. In other words, it is a configuration whose control
state is $q_{\text{acc}}$. A halting configuration is a configuration whose control state is either
$q_{\text{acc}}$ or $q_{\text{rej}}$, i.e., it is a member of the set $(q_{\text{acc}}, q_{\text{rej}}) \times \mathbb{N} \times (\Gamma^{\ast} \{ * \} \Gamma^{\ast} \sqcup^{\omega})^{k}$.

Having defined configurations, we can formally define how configurations change
in a single step of the Turing machine. We begin by defining a function that updates
the work tape. For a work tape $u \ast \sqcup \sqcup^{\omega}$, $\text{upd}(u \ast \sqcup \sqcup^{\omega}, b, d)$ is the resulting work
tape when $b$ is written and the head is moved in direction $d$. This can be formally defined as

$$\text{upd}(u \ast \sqcup \sqcup^{\omega}, b, d) = \begin{cases} \text{if } d = +1 \text{ and } v = \varepsilon & ub \ast \sqcup \sqcup^{\omega} \\ \text{if } d = +1 \text{ and } v = cv' & ub \ast cv' \sqcup^{\omega} \\ \text{if } d = -1 \text{ and } u = u'c & u' \ast cv \sqcup^{\omega} \end{cases}$$

Recall also that for a finite string $w \in \Gamma^{\ast}$, $w[i]$ denotes the $i$th symbol in the string. We can extend this notion to tape contents that are sequences of the form $w \sqcup^{\omega}$ as follows.

$$w \sqcup^{\omega}[i] = \begin{cases} w[i] & \text{if } i < |w| \\ \sqcup & \text{otherwise} \end{cases}$$

**Definition A.3 (Computation Step)**

Consider configurations $c_1 = (q_1, i_1, u_1 \ast a_1 v_1 \ldots \ast a_k v_k)$ and $c_2 = (q_2, i_2, u_2 \ast a_1 v_1 \ldots \ast a_k v_k)$ of Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \sqcup, \mathrel{\triangleleft})$. Let the input string be $w$. We say $c_1 \xrightarrow{\delta} c_2$ (machine $M$ moves from configuration $c_1$ to $c_2$ in one step and writes $o$ on the output tape) if the following conditions hold. Let $\delta(q_1, w \sqcup^{\omega}[i_1], a_1, \ldots, a_k) = (p, d_0, b_1, d_1, \ldots, b_k, d_k)$. Then,

- $q_2 = p$,
- $i_2 = i_1 + d_1$,
- for each $i$, $i_i = \text{upd}(u_i \ast a_i v_i, b_i, d_i)$

When the output symbol written during a step is not important, we will write $c_1 \leftrightarrow c_2$ to indicate a step from $c_1$ to $c_2$.

Having defined how the configuration of a Turing machine changes in each step, we can define the result of a computation on an input.

**Definition A.4 (Computation)**

A computation of Turing machine $M$ on input $w$, is a sequence of configurations $c_1, c_2, \ldots, c_m$ such that $c_1$ is the initial configuration of $M$, and for each $i$, $c_i \leftrightarrow c_{i+1}$.

**Definition A.5 (Acceptance)**

An input $w$ is accepted by Turing machine $M$ if there is a computation $c_1, c_2, \ldots, c_m$ such that $c_m$ is an accepting configuration.
The language recognized/accepted by $M$ is $L(M) = \{ w \mid w$ is accepted by $M \}$. We say that a language $A \subseteq \Sigma^*$ is accepted/recognized by $M$ if $L(M) = A$.

**Definition A.6 (Halting)**

A Turing machine $M$ is said *halt* on input $w$ if there is a computation $c_1, c_2, \ldots, c_m$ such that $c_m$ is a halting configuration.

The Turing machine model we introduced with an output tape can be used to compute (partial) functions as follows.

**Definition A.7 (Function Computation)**

The partial function computed by a Turing machine $M$, denoted $f_M$, is as follows. If on input $w$, $M$ has a halting computation $c_1 \xrightarrow{\alpha_1} c_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{m-1}} c_m$ then $f_M(w)$ is defined and equal to $\alpha_1 \alpha_2 \cdots \alpha_{m-1}$. On inputs $w$ such that $M$ does not halt, $f_M(w)$ is undefined.

We say that a (partial) function $g$ is computable if there is a Turing machine $M$ such that for every $w$, $g(w)$ is defined if and only if $f_M(w)$ is defined, and whenever $g(w)$ is defined, $g(w) = f_M(w)$.

Most of the time we will be considering Turing machines that accept or recognize languages, rather than those that compute functions. In this context, the symbols written on the output tape don’t matter, and so we will often ignore the output tape when describing transitions and computations of such machines.

### A.2 Church-Turing Thesis

The Turing machine model introduced in the previous section, is a canonical model to capture mechanical computation. The Church-Turing thesis embodies this statement by saying that anything solvable using a mechanical procedure can be solved using a Turing machine. Our belief in the Church-Turing thesis is based on decades of research in alternate models of computation, which all have turned out to be computationally equivalent to Turing machines. Some of these models include the following.

- Non-Turing machine models: Random Access Machines, $\lambda$-calculus, type 0 grammars, first-order reasoning, $\pi$ calculus, …
- Enhanced Turing machine models: Turing machines with multiple 2-way infinite tapes, nondeterministic Turing machines, probabilistic Turing machines, quantum Turing machines, …
- Restricted Turing machine models: Single tape Turing machines, Queue machines, 2-stack machines, 2-counter machines, …

We will choose to highlight two of these results, that will play a role in our future discussions. The first is the observation that a one work tape Turing machine is computationally as powerful as the multi-work tape model introduced in Definition A.1.
Theorem A.8 For any $k$ work tape Turing machine $M$, there is a Turing machine with a single work tape single($M$) such that $L(M) = L(\text{single}(M))$ and $f_M = f_{\text{single}(M)}$.

Proof of Theorem A.8 can be found in any standard textbook and its precise details are skipped. The idea behind the proof is as follows. The single work tape machine single($M$) will simulate the steps of the $k$-work tape machine $M$ on any input. But in order to simulate $M$, single($M$) needs to keep track of $M$’s configuration at each step. That means keeping track of $M$’s state, its work tape contents, and its tape head. This single($M$) accomplishes by storing $M$’s state in its own state, and the contents of all $k$ work tapes of $M$ (including the head positions) on the single work tape of single($M$). In general, cell $i$ of the single work tape, stores cell $(i \div k) + 1$ of tape $i$ mod $k$; here $i \div m$ denotes the quotient when $i$ is divided by $m$ and $i$ mod $m$ denotes the remainder. Then to simulate a single step of $M$, single($M$) will make multiple passes over its single work tape, to first identify the symbols on each tape read by $M$ to determine the transition to take, and then update the contents of the tape according to the transition.

The second result relates to the nondeterministic Turing machines. The Turing machine model introduced in Definition A.1 is deterministic, in the sense that at any given time during the computation of the machine, there is at most one possible transition the machine can take. Nondeterminism, on the other hand, is the computational paradigm where the computing device, at each step, may have multiple possible transitions to choose from. As a consequence, on a given input the machine may have multiple computations, and the machine is said to accept an input, if any one of these computations leads to an accepting configuration. Formally, we can define a nondeterministic Turing machine as follows.

Definition A.9 A nondeterministic Turing machine with $k$ work tapes (and one input tape) is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \sqcup, \triangleright)$, where $Q, \Sigma, \Gamma, q_0, q_{\text{acc}}, q_{\text{rej}}, \sqcup, \triangleright$ are just like that for deterministic Turing machine, and

$$\delta : (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma^{k+1} \rightarrow 2^{Q \times \{-1,+1\} \times (\Gamma \times \{-1,+1\})}$$

is the transition function. The transition function, given current state and symbols read on the input and work tapes, returns a set of possible next states, direction to move the input head, and symbols to be written and direction to move the head in for each work tape.

The definition of configurations, initial configuration, accepting and halting configurations is the same as in Definition A.2. The definitions of computation step (Definition A.3), computation (Definition A.4), and acceptance and language recognized (Definition A.5) are also the same. Hence we skip defining these formally.

---

3 For partial functions $f$ and $g$, we write $f = g$ to indicate that $f$ and $g$ have the same domains (i.e., they are defined for exactly the same elements), and further when $f(x)$ is defined, $f(x) = g(x)$.

4 We assume there is no output tape for a nondeterministic Turing machine since such machines are used for function computation.
Every deterministic Turing machine is a special kind of nondeterministic machine, namely, one which has the property that at each time step there is at most one transition enabled. One of the important results concerning nondeterministic Turing machines is that the converse is also true, i.e., nondeterministic Turing machines are not more powerful than deterministic Turing machines.

**Theorem A.10** For every nondeterministic Turing machine \( N \), there is a deterministic Turing machine \( \text{det}(N) \) such that \( L(N) = L(\text{det}(N)) \).

A detailed proof of Theorem A.10 is skipped. It can be found in any standard textbook in theory of computation. The broad idea behind the result is the observation that once the length of computation, and the nondeterministic choices at each step are fixed, a deterministic machine can simulate \( N \) for that length, on those choices. Thus, the deterministic Turing machine \( \text{det}(N) \) simulates \( N \) for increasingly longer computations, and for each length, \( \text{det}(N) \) will cycle through all possible nondeterministic choices at each step. If any of these computations is accepting for \( N \), then \( \text{det}(N) \) will halt and accept.

### A.3 Recursive and Recursively Enumerable Languages

The Church-Turing thesis establishes the canonicity of the Turing machine as a model of mechanical computation. The collection of problems solvable on Turing machines is, therefore, worthy of study. Recall that when a Turing machine \( M \) is run on an input string \( w \) there are 3 possible outcomes — \( M \) may (halt and) accept \( w \), \( M \) may (halt and) reject \( w \), or \( M \) may not halt on \( w \) (and therefore not accept). Depending on how a Turing machine behaves we can define two different classes of problems solvable on a Turing machine.

**Definition A.11** A language \( A \) is recursively enumerable/semi-decidable if there is a Turing machine \( M \) such that \( A = L(M) \).

A language \( A \) is recursive/decidable if there is a Turing machine \( M \) that halts on all inputs and \( A = L(M) \).

Observe that when a problem \( A \) is recursive/decidable, it has a special algorithm that solves it and in addition always halts, i.e., on inputs not in \( A \), this algorithm explicitly rejects. Thus, by definition, every recursive language is also recursively enumerable.

**Proposition A.12** If \( A \) recursive then \( A \) is recursively enumerable.

We will denote the collection of recursive languages as \( \text{REC} \) and the collection of all recursively enumerable languages as \( \text{RE} \); thus, Proposition A.12 can be seen as saying that \( \text{REC} \subseteq \text{RE} \). The collection of recursive and recursively enumerable languages enjoy some closure properties that are worth recalling.

**Theorem A.13** \( \text{REC} \) is closed under all Boolean operations while \( \text{RE} \) is closed under monotone Boolean operations. That is,
If \( A, B \in \text{RE} \), then \( A \cup B \) and \( A \cap B \) are also in \( \text{RE} \).

If \( A, B \in \text{REC} \), then \( \overline{A} \), \( A \cup B \), and \( A \cap B \) are all in \( \text{REC} \).

**Proof** We will focus on the two most interesting observations in Theorem A.13; the rest we leave as an exercise for the reader. The first observation we will prove is the closure of \( \text{RE} \) under union. Let us assume \( M_A \) and \( M_B \) are Turing machines recognizing \( A \) and \( B \), respectively. The computational problem \( A \cup B \) asks one to determine if a given input string \( w \) belongs to either \( A \) or \( B \). We could determine membership in \( A \) and \( B \) by running \( M_A \) and \( M_B \), respectively, but we need to be careful about how we run \( M_A \) and \( M_B \). Suppose we choose to first run \( M_A \) on \( w \) and then run \( M_B \) on \( w \), then we could run into problems. For example, consider the situation where \( M_A \) does not halt on \( w \), but \( w \in \overline{A} \). Then, running \( M_A \) followed by \( M_B \) will never run \( M_B \) and therefore never accept, even though \( w \in A \cup B \). Switching the order of running \( M_A \) and \( M_B \) also does not help. What one needs to instead do is, to run \( M_A \) and \( M_B \) simultaneously on \( w \). How does one \( M_A \) and \( M_B \) at the same time? There are many ways to achieve this. One way is to initially run one step of \( M_A \) and then one step of \( M_B \) on \( w \) from the initial configuration. If either them accept, the algorithm for \( A \cup B \) accepts. If not, it will run \( M_A \) for two steps, and \( M_B \) for two steps, again starting from the respective initial configurations. Again, the algorithm for \( A \cup B \) accepts if either simulation accepts. If not the computations of \( M_A \) and \( M_B \) are increased by one more step, and this process continues, until at some point one of them accepts.

The second result we would like to focus on is the observation that \( \text{REC} \) is closed under complementation. Let \( A \in \text{REC} \) and let \( M \) be a Turing machine that halts on all inputs and \( L(M) = A \). The algorithm \( \overline{M} \) for \( \overline{A} \), runs \( M \) on input \( w \), and if \( M \) accepts it rejects and if \( M \) rejects then it accepts. Notice that \( L(\overline{M}) = \overline{A} \) only because \( M \) halts on all inputs — if \( M \) does on halt on (say) \( w \), then \( w \in A \) but \( \overline{M} \) would never accept \( w \! \) 

The following theorem is a useful way to prove that a problem is decidable.

**Theorem A.14** \( A \) is recursive if and only if \( A \) and \( \overline{A} \) are recursively enumerable.

**Proof** If \( A \in \text{REC} \) then \( \overline{A} \in \text{REC} \) by Theorem A.13. Then both \( A \) and \( \overline{A} \) are recursively enumerable by Proposition A.12.

Conversely, suppose \( A \) and \( \overline{A} \) are recognized by \( M_A \) and \( M_{\overline{A}} \) respectively. The recursively algorithm \( M \) for \( A \), on a given input \( w \), will run both \( M_A \) and \( M_{\overline{A}} \) simultaneously (as in the proof of Theorem A.13), and accept if either \( M_A \) accepts or \( M_{\overline{A}} \) rejects. Notice, that any given input \( w \) belongs to either \( A \) or \( \overline{A} \), and therefore at least one out of \( M_A \) and \( M_{\overline{A}} \) is guaranteed to halt on each input. Therefore \( M \) will always halt.

**Encodings.**

Every object (graphs, programs, Turing machines, etc.) can be encoded as a binary string. The details of the encoding scheme itself are not important, but it should be
simple enough that the data associated with the object should be easily recoverable by reading the binary encoding. For example, one should be able to reconstruct the vertices and edges of a graph from its encoding, or one should be able to reconstruct the states, transitions, etc. of a Turing machine from its encoding. For a list of objects $O_1, O_2, \ldots, O_n$, we will use $\langle O_1, O_2, \ldots, O_n \rangle$ to denote their binary encoding. In particular, for a Turing machine $M$, $\langle M \rangle$ is its encoding as binary string. Conversely, for a binary string $x$, $M_x$ denotes the Turing machine whose encoding is the string $x$.

Once we establish an encoding scheme, we can construct a *Universal Turing machine*, which is an *interpreter* that given an encoding of a Turing machine $M$ and an input $w$, can simulate the execution of $M$ on the input string $w$. This is an extremely important observation that establishes the recursive enumerability of the membership problem for Turing machines.

**Theorem A.15** There is a Turing machine $U$ (called the universal Turing machine) that recognizes the language $MP = \{ \langle M, w \rangle \mid w \in L(M) \}$. In other words, $MP \in \text{RE}$.

Not every decision problem/language is recursively enumerable. Using Cantor’s diagonalization technique, one can establish the following result.

**Theorem A.16** The language $\overline{K} = \{ x \mid x \notin L(M_x) \}$ is not recursively enumerable.

**Proof** The proof of Theorem A.16 relies on a diagonalization argument to show that the language of every Turing machine differs from $\overline{K}$, and therefore $\overline{K}$ is not recursively enumerable.

Consider an arbitrary Turing machine $M_x$ whose encoding as a binary string is $x$. We will show that $L(M_x) \neq \overline{K}$, thereby proving the theorem. Observe that if $x \in L(M_x)$ then by definition $x \notin \overline{K}$ and if $x \notin L(M_x)$ then again by definition $x \in \overline{K}$. Therefore $x \in (\overline{K} \setminus L(M_x)) \cup (L(M_x) \setminus \overline{K}) \neq \emptyset$.

### A.4 Reductions

Theorem A.16 is the first result that establishes that there are problems that are computationally difficult. Further results on the computational hardness of problems are usually established using the notion of *reductions*. Reductions demonstrate how one problem can be converted into another in such a way that a solution to the second problem can be used to solve the first. Formally, it is defined as follows.

**Definition A.17** A *(many-one/mapping)* reduction from $A$ to $B$ is a computable (total) function $f : \Sigma^* \to \Sigma^*$ such that for any input string $w$,

$$w \in A \text{ if and only if } f(w) \in B$$

In this case, we say $A$ is *(many-one/mapping)* reducible to $B$ and we denote it by $A \leq_m B$. 
Since many-one/mapping reductions are the only form of reduction we will study, we will drop the adjective “many-one” and “mapping” and simply call these reductions. Let us look at a couple of examples of reductions.

Example A.18 Let us consider the complement of $\text{MP}$, i.e., $\overline{\text{MP}} = \{\langle M, w \rangle | w \notin \text{L}(M)\}$. One can show that $\overline{K} \leq_m \overline{\text{MP}}$ as follows. The reduction $f$ is the following function: $f(x) = \langle M_x, x \rangle$.

To prove that $f$ is a reduction, we need to argue two things. First that $f$ is computable, i.e., we need to come up with a Turing machine $M_f$ that always halts and produces the string $f(x)$ on input $x$. In this example, to construct $f(x)$, we simply need to “copy” the string $x$ which clearly is a computable function. Second we need to argue that $x \in \overline{K}$ iff $f(x) \in \overline{\text{MP}}$. This can be argued as follows: $x \in \overline{K}$ iff $x \notin \text{L}(M_x)$ (definition of $\overline{K}$) iff $\langle M_x, x \rangle \in \overline{\text{MP}}$ (definition of $\overline{\text{MP}}$) iff $f(x) \in \overline{\text{MP}}$ (definition of $f$).

Example A.19 Consider the problem

$$\overline{\text{HP}} = \{\langle M, w \rangle | M \text{ does not halt on } w\}.$$  

We will prove that $\overline{K} \leq_m \overline{\text{HP}}$.

Given a binary string $x$, let us consider the following program $H_x$.

```plaintext
H_x(w)
    result = M_x(x)
    if (result = accept)
        return accept (* on input w *)
    else
        while true do
```

In other words, the program $H_x$ on input $w$, ignores its input and runs the program $M_x$ on $x$. If $M_x$ halts and accepts $x$ then $H_x$ halts and accepts $w$. Otherwise, $H_x$ does not halt. Thus, the program $H_x$ halts on some (all) inputs if and only if $x \in \text{L}(M_x)$.

Let us now describe the reduction from $\overline{K}$ to $\overline{\text{HP}}$: $f(x) = \langle H_x, x \rangle$. Observe first that $f$ satisfies the properties of a reduction because $x \in \overline{K}$ iff $x \notin \text{L}(M_x)$ iff $H_x$ does not halt on $x$ (and all input strings) iff $\langle H_x, x \rangle \in \overline{\text{HP}}$. To establish that $f$ is a reduction, we also need to argue that $f$ is computable. On input string $x$, we need a program that produces the source code for $H_x$ (given above) and copies the string $x$ after the source code. This is clearly computable.

Reducions are a way for one to compare the computational difficulty of problems — if $A$ reduces to $B$ then $A$ is at most as difficult as $B$, or $B$ is at least as difficult as $A$. This is formally captured in the following proposition.

**Theorem A.20** If $A \leq_m B$ and $B$ is recursively enumerable (recursive) then $A$ is recursively enumerable (recursive).
Fig. A.2 Schematic argument for Theorem A.20.

**Proof** Let $f$ be a reduction from $A$ to $B$ that is computed by Turing machine $M_f$, and let $M_B$ be a Turing machine that recognizes $B$. The algorithm for $A$ is schematically shown in Fig. A.2 — on input $w$, compute $f(w)$ using $M_f$ and run $M_B$ on $f(w)$. Notice that this algorithm always halts if $M_B$ always halts. Thus, if $B$ is recursive then $A$ is also recursive. □

Theorem A.20 can be seen to informally say “if $A$ reduces to $B$ and $B$ is computationally easy then $A$ is computationally easy”. It is often used in the contrapositive sense and it is useful to explicitly state this observation.

**Corollary A.21** If $A \leq_m B$ and $A$ is not recursively enumerable (undecidable) then $B$ is not recursively enumerable (undecidable).

We can use the above corollary to argue the computational hardness of some problems.

**Theorem A.22** $\overline{M_P}$ is not recursively enumerable. Therefore, $M_P$ is undecidable.

**Proof** Example A.18 establishes that $\overline{K} \leq_m \overline{M_P}$. Together with Theorem A.16 and Corollary A.21, we can conclude that $\overline{M_P}$ is not recursively enumerable. Finally, since $\overline{M_P}$ is not recursively enumerable, Theorem A.14 establishes that $M_P$ is not decidable/recursive. □

Since $M_P \in \text{RE}$ (Theorem A.15) and $\overline{M_P} \notin \text{RE}$ (Theorem A.22), we have a witness to the fact that $\text{RE}$ is not closed under complementation. Just like Theorem A.22, we could establish similar properties for the halting problem.

**Theorem A.23** $\overline{H_P}$ is not recursively enumerable. Therefore, $H_P = \{\langle M, w \rangle \mid M \text{ halts on } w \}$ is undecidable.

**Proof** Follows from Example A.19 and the argument in the proof of Theorem A.22. □

Reductions are transitive and hence a pre-order; thus, the use of $\leq$ to denote them is justified.

**Theorem A.24** The following properties hold for reductions.
• If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$.
• If $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.

**Proof** If $f$ is a reduction from $A$ to $B$, then one can argue that $f$ is also a reduction from $\overline{A}$ to $\overline{B}$. And, if $f$ is a reduction from $A$ to $B$ and $g$ a reduction from $B$ to $C$ then $g \circ f$ is a reduction from $A$ to $C$. Establishing these observations to prove the theorem is left as an exercise. □

Having found a lens to compare the computational difficulty of two problems (namely, reductions), one can use them to argue that a problem is at least as difficult as a whole collection of problems, or something is the “hardest” problem in a collection. This leads us to notions of hardness and completeness.

**Definition A.25** A language $A$ is RE-hard if for every $B \in \text{RE}, B \leq_m A$.

A language $A$ is RE-complete if $A$ is RE-hard and $A \in \text{RE}$.

Thus, an RE-complete problem is the hardest problem that is recursively enumerable, while an RE-hard problem is something that is at least as hard as any other RE problem. Are there examples of such problems? It turns out that MP, HP, and K are all RE-complete. We establish this for MP in the following theorem.

**Theorem A.26** MP is RE-complete.

**Proof** Membership in RE has been established in Theorem A.15. So all we need to prove is the hardness. Let $B$ be any recursively enumerable language, and let $M$ be a Turing machine recognizing $B$. The reduction from $B$ to MP is as follows: $f(w) = \langle M, w \rangle$. It is easy to see that $w \in B$ iff $w \in L(M)$ (since $M$ recognizes $B$) iff $\langle M, w \rangle \in MP$ (definition of MP) iff $f(w) \in MP$ (definition of $f$). It is also easy to see that $f$ is computable — in order to compute $f(w)$, all we need to do is prepend the source code of $M$.

Establishing RE-hardness of a problem is sufficient to guarantee it’s undecidability.

**Theorem A.27** If $A$ is RE-hard then $A$ is undecidable.

**Proof** If $A$ is RE-hard then since MP $\in \text{RE}$, we have MP $\leq_m A$. Since MP is undecidable (Theorem A.22), by properties of a reduction (Corollary A.21) $A$ is undecidable. □

**A.5 Complexity Classes**

Computational resources needed to solve a problem depend on the size of the input instance. For example, it is clearly easier to compute the sum of two one digit numbers as opposed to adding two 15 digit numbers. The resource requirements of an algorithm/Turing machine are measured as a function of the input size. We will only study time and space as computational resources in this presentation. We
begin by defining time bounded and space bounded Turing machines, which are defi

definitions apply to both deterministic and nondeterministic machines.

**Definition A.28** A (deterministic/nondeterministic) Turing machine $M$ is said to run
in time $T(n)$ if on any input $u$, all computations of $M$ on $u$ take at most $T(|u|)$ steps;
here $|u|$ refers to the length of input $u$.

A (deterministic/nondeterministic) Turing machine $M$ is said to use space $S(n)$
if on any input $u$, all computations of $M$ on $u$ use at most $S(|u|)$ work tape cells. In
this context, a work tape cell is said to be used if it is written to at least once during
the computation. Notice that, if a work tape cell is written multiple times during
a computation, it counts as only one cell when measuring the space requirements;
thus, work tape cells can be reused without adding to the space bounds.

It is worth examining Definition A.28 carefully. Our requirement for a Turing
machine running within some time or space bound applies to all computations,
whether they are accepting or not. Notice also that the definition is the same for
both deterministic and nondeterministic models — in a deterministic machine the
unique computation on a given input must satisfy the resource bounds, and in a
nondeterministic machine, all computations on the input must satisfy the bounds.
In particular, if a Turing machine (deterministic or nondeterministic) runs within a
time bound, then it halts in every computation of every input.

Having defined time and space bounded machines, we can define the basic com-
plexity classes which are collections of (decision) problems that can be solved within
certain time and space bounds.

**Definition A.29** We define the following basic complexity classes.

- A language $A \in \text{DTIME}(T(n))$ iff there is a deterministic Turing machine that
  runs in time $T(n)$ such that $A = L(M)$.
- A language $A \in \text{NTIME}(T(n))$ iff there is a nondeterministic Turing machine that
  runs in time $T(n)$ such that $A = L(M)$.
- A language $A \in \text{DSPACE}(S(n))$ iff there is a deterministic Turing machine that
  uses space $S(n)$ such that $A = L(M)$.
- A language $A \in \text{NSPACE}(S(n))$ iff there is a nondeterministic Turing machine
  that uses space $S(n)$ such that $A = L(M)$.

Our computational model of Turing machines, and our definitions of time and
space bounded computations are robust with respect to constant factors. This obser-
vation is captured by two central results in theoretical computer science, namely, the
linear speedup and compression theorems. It says that one can always improve the
running time or space requirements for solving a problem by a constant factor.

**Theorem A.30 (Linear Speedup)**

If $A \in \text{DTIME}(T(n))$ (or $A \in \text{NTIME}(T(n))$) and $c > 0$ is any constant, then
$A \in \text{DTIME}(cT(n) + n)$ ($A \in \text{NTIME}(cT(n) + n)$).
Proof (Sketch) Let $A = L(M)$. We will describe a machine $M'$ which will simulate $k$ steps of $M$ in 8 steps; if $k > \frac{8}{3}$, we will get the desired result. $M'$ will have one more work tape, a much larger tape alphabet, and control states than $M$.

- $M'$ copies the input onto the additional work tape in compressed form: $k$ successive symbols of $M$ will be represented by one symbol in $M'$. Time taken is $n$. $M'$ will maintain $M$’s work tape contents in compressed form on the second work tape as well.
- $M'$ uses the additional work tape as “input tape”. The head positions of $M$, within the $k$ symbols represented by current cells, is stored in finite control.

One basic move of $M'$ (consisting of 8 steps), will simulate $k$ steps of $M$ as follows.

- At the start of basic move, $M'$ moves its tape heads one cell left, two cells right and one cell left, storing the symbols read in the finite control. Now, $M'$ knows all symbols within the radius of $k$ cells of any of $M$’s tape heads. This takes 4 steps.
- Based on the transition function of $M$, $M'$ can compute the effect of the next $k$ steps of $M$.
- Using any additional (at most) 4 steps, $M'$ updates the contents of its tapes as a result of the $k$ steps, and moves the heads appropriately. □

Theorem A.31 (Linear Compression)

If $A \in \text{DSPACE}(S(n))$ (or $A \in \text{NSPACE}(S(n))$) and $c > 0$ is any constant then $A \in \text{DSPACE}(cS(n))$ (or $A \in \text{NSPACE}(cS(n))$).

Proof Increase the tape alphabet size and store work tape contents in compressed form as in Theorem A.30. □

Theorems A.30 and A.31 suggest that when analyzing the time and space requirements of an algorithm we can ignore constant terms. This leads to the use of the order notation.

Definition A.32 Consider functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

- $f(n) = O(g(n))$ if there are constants $c, n_0$ such that for $n > n_0$, $f(n) \leq cg(n)$. $g(n)$ is an asymptotic upper bound.
- $f(n) = \Omega(g(n))$ if there are constants $c, n_0$ such that for $n > n_0$, $f(n) \geq cg(n)$. $g(n)$ is an asymptotic lower bound.

The complexity classes identified in Definition A.29 are a very fine classification of problems. They include complexity classes whose classification of problems is sensitive to our use of Turing machines as a model of computation. Ideally we would like to study complexity classes such that if a problem is classified in a certain class then that classification should be “platform independent”. That is, whether we choose to study complexity on Turing machines or Random Access Machines, our observations should still hold. They should also be invariant under small changes to the Turing machine model, like changing the number of work tapes, alphabet, nature
of the tapes, etc. There is a strengthening of the Church-Turing thesis, called the \textit{invariance thesis} articulated by Church, that underlies our belief in the robustness of the Turing machine model, subject to small changes in the time and space bounds. It says that

Any effective, mechanistic procedure can be simulated on a Turing machine using the same space (if space is \(\geq \log n\)) and only a polynomial slowdown (if time \(\geq n\))

In addition to the requirement that complexity classes be robust to changes to the computational platform, we would like the classes to be closed under function composition — making function/procedure calls to solve sub-problems is a standard algorithmic tool, and we would like the complexity to remain the same as long as the sub-problems being solved are equally simple. Finally, we would like our complexity classes to capture natural, “interesting”, real-world problems. For these reasons, we typically study the following complexity classes that provide a coarser classification of problems than that provided in Definition A.29.

\textbf{Definition A.33} Commonly studied complexity classes are the following.

\begin{align*}
L &= \text{DSPACE}(\log n) & NL &= \text{NSPACE}(\log n) \\
P &= \bigcup_k \text{DTIME}(n^k) & NP &= \bigcup_k \text{NTIME}(n^k) \\
PSPACE &= \bigcup_k \text{DSPACE}(n^k) & \text{NPSPACE} &= \bigcup_k \text{NSPACE}(n^k) \\
\text{EXP} &= \bigcup_k \text{DTIME}(2^{n^k}) & \text{NEXP} &= \bigcup_k \text{NTIME}(2^{n^k})
\end{align*}

In addition to the above classes, for any class \(C\), \(\text{co}C = \{A \mid \overline{A} \in C\}\). Please note that \(\text{co}C\) is \textit{not} the complement of \(C\) but instead is the collection of problems whose complement is in \(C\).

\section*{A.6 Relationship between Complexity Classes}

We begin by relating time and space complexity classes.

\textbf{Theorem A.34} \(\text{DTIME}(T(n)) \subseteq \text{DSPACE}(T(n))\) and \(\text{NTIME}(T(n)) \subseteq \text{NSPACE}(T(n))\)

\textbf{Proof} A Turing machine can scan at most one new work tape cell in any step. Therefore, the number of work tape cells used during a computation cannot be more than the number of steps. \(\square\)

\textbf{Theorem A.35} \(\text{DSPACE}(S(n)) \subseteq \text{DTIME}(n \cdot 2^{O(S(n))})\) and \(\text{NSPACE}(S(n)) \subseteq \text{NTIME}(n \cdot 2^{O(S(n))})\). In particular, when \(S(n) \geq \log n\), we have \(\text{DSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})\) and \(\text{NSPACE}(S(n)) \subseteq \text{NTIME}(2^{O(S(n))})\).

We will skip giving a direct proof of Theorem A.35. It will follow from Theorem A.37 and Theorem A.38. An immediate consequence of Theorems A.34 and A.35 are the following relationships between the complexity classes.
Corollary A.36

\[ \mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \]
\[ \mathsf{NL} \subseteq \mathsf{NP} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{NEXP} \]

We will establish relationships between deterministic and nondeterministic complexity classes.

Theorem A.37

\[ \mathsf{DTIME}(T(n)) \subseteq \mathsf{NTIME}(T(n)) \text{ and } \mathsf{DSPACE}(S(n)) \subseteq \mathsf{NSPACE}(S(n)). \]

**Proof** This follows from the fact that, by definition, every deterministic Turing machine is a special nondeterministic Turing machine, namely, those that have exactly one transition enabled from every non-halting configuration. \(\Box\)

Nondeterministic complexity class can also be related to deterministic complexity classes. In fact, we now prove a result that subsumes the containment results for nondeterministic classes established in Theorems A.34 and A.35.

Theorem A.38

\[ \mathsf{NTIME}(T(n)) \subseteq \mathsf{DSPACE}(T(n)) \text{ and } \mathsf{NSPACE}(S(n)) \subseteq \mathsf{DTIME}(n^{O(S(n))}). \]

**Proof** Let us begin by proving the first inclusion. Consider \(A \in \mathsf{NTIME}(T(n))\) and let \(M\) be \(T(n)\)-time bounded nondeterministic machine recognizing \(A\). On any input \(w\) of length \(n\), the computations of \(M\) can be organized as a tree, and since \(M\) runs in time \(T(n)\), this tree has height \(T(n)\). Now the deterministic algorithm \(D\) to solve \(A\) will perform a depth first search (DFS) on this computation tree of \(M\), constructing this tree as it is explored, and accepting if some node in this computation tree corresponds to an accepting configuration. The space needed by \(D\) to perform this DFS is the memory needed to store the call stack. The stack during a DFS keeps track of the path being currently explored in the tree to enable backtracking. Since the computation tree of \(M\) is of height \(T(n)\), the height of the call stack is also \(T(n)\). A naïve implementation of the DFS algorithm will store the sequence of tree vertices on the current path; since in this case each vertex is a configuration of \(M\), these can be represented by strings of length \(T(n)\) (as work tape cells cannot exceed \(T(n)\) as in Theorem A.34). This gives us a space bound of \(T(n)^2\) for algorithm \(D\). However, instead of storing the actual configurations in the computation being currently explored, \(D\) can just store the sequence of nondeterministic choices made by \(M\) in the current computation. With this information about the nondeterministic choices, \(D\) can reconstruct the configuration at the end of a sequence of steps, by resimulating \(M\) from the beginning — this increases the running time of \(D\), but reduces the space requirements of \(D\) which is what we care about for this result. If \(M\) has \(k\) choices at each step, the stack of \(D\) during DFS is simply a \(k\)-ary string of length \(\leq T(n)\), which means that \(D\) is \(T(n)\)-space bounded.

For the second result, let us consider \(A \in \mathsf{NSPACE}(S(n))\) and a nondeterministic Turing machine \(M\) that recognizes \(A\) in \(S(n)\) space. On a given input \(w\) of the length \(n\), it is useful to define the notion of a configuration graph of \(M\). The configuration graph is a directed graph that has as vertices, configurations of \(M\), and has an edge from \(c_1\) to \(c_2\), if \(M\) can move from configuration \(c_1\) to configuration \(c_2\) in one step.
Fig. A.3 Relationship between Complexity Classes. → indicates containment, though whether it is strict is unknown.

given input $w$. Observe that $M$ accepts $w$ if an accepting configuration is reachable from the initial configuration in this configuration graph. Notice also that since $M$ is $S(n)$-space bounded, the total number of vertices in this graph is $\leq 2^{O(S(n))}$ (see proof of Theorem A.35). Now we can run our favorite graph search algorithm (depth first search or breadth first search) on this configuration graph to see if an accepting configuration is reachable; the graph will be constructed on-the-fly as it is being explored. Such an algorithm (which deterministic), takes time that is linear in the size of the graph, which gives us a $2^{O(S(n))}$ deterministic algorithm for $A$. 

Our new observations relating deterministic and nondeterministic complexity classes gives us the following relationships.

**Corollary A.39**

\[ L \subseteq NL \subseteq P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{NPSPACE} \subseteq \text{EXP} \subseteq \text{NEXP} \]

An important result due to Savitch, relates nondeterministic and deterministic space complexity classes. The interested reader can find its proof in textbooks like [?].

**Theorem A.40 (Savitch)**

For $S(n) \geq \log n$, $\text{NSPACE}(S(n)) \subseteq \text{DSPACE}(S(n)^2)$. In particular, this means that $\text{PSPACE} = \text{NPSPACE}$. 

Putting all our observations together we get the relationships shown in Fig. A.3. It is worth observing that for any deterministic complexity class $C$, $C = \text{coC}$; this is because an algorithm for $\overline{A}$ is to run the deterministic algorithm for $A$ and flip the final answer. Thus, $P = \text{coP}$. Further, from Theorem A.37, we have $\text{coP} \subseteq \text{coNP}$, giving us the containment $P \subseteq \text{coNP}$. Finally, due the space hierarchy theorem, we also know that $L \neq \text{PSPACE}$ and $NL \neq \text{PSPACE}$, and from the time hierarchy theorem, we know that $P \neq \text{EXP}$ and $NP \neq \text{NEXP}$; the hierarchy theorems are beyond the scope of this brief primer.

A.7 P and NP

*Cobham-Edmonds Thesis*, named after Alan Cobham and Jack Edmonds, asserts that the only computational problems that have “efficient” or “feasible” algorithmic
solutions are those that belong to P. In other words, P is the collection of *tractable* computational problems. There are many features of P that justify this view.

- The invariance thesis suggests that any problem in P can be solved in polynomial time on *any* reasonable computational model. Thus, the statement of a problem being efficiently computable is platform independent.
- Most encodings of an input structure are polynomially related in terms of their length. Thus, if a problem is in P for one encoding, it will be in P even if input instances are encoded in a different manner. Therefore, P is insensitive to problem encodings.
- Most natural problems in P have algorithms whose running time is bounded by a low-order polynomial. Thus, their running times are likely to be low for most problem instances.
- The asymptotic growth of polynomials is moderate when compared to the astounding growth of exponential functions. Thus, problems in P are likely to be feasibly solved even on large problem instances.

The crux of the Cobham-Edmonds thesis is that for a problem to be solvable in practice, it should have a polynomial time algorithm. Therefore, much effort in the past 50 years has been devoted to understanding the class of problems in P. In particular, can we prove that the complexity classes containing P in Fig. A.3, like NP and PSPACE, also have efficient solutions, i.e., are contained in P? Or can we say for certain that some problems in NP and PSPACE cannot be solved efficiently?

### A.7.1 Alternate characterization of NP

We defined NP as the collection of problems that can be solved in polynomial time on a nondeterministic Turing machine. In this section, we will give an alternate definition, namely, as those problems that are efficiently “verifiable”.

**Definition A.41** A language $A$ is *polynomially verifiable* if there is a $k \in \mathbb{N}$ and a deterministic Turing machine $V$ such that

$$A = \{w \mid \exists p. \ V \text{ accepts } \langle w, p \rangle\}$$

and $V$ takes at most $|w|^k$ steps on input $\langle w, p \rangle$, i.e., $V$ running time is *independent* of the length of $p$. Here $V$ is called a *verifier* for $A$, and for $w \in A$, the strings $p$ such that $\langle w, p \rangle$ is accepted by $V$ are called a *proof* of $w$ (with respect to $V$).

The notion of a language $A$ being polynomially verifiable says that when a string $w \in A$, there is a *proof* $p$ (maybe even more than one) such that $w$ augmented with $p$ “convinces” $V$, i.e., causes $V$ to accept. However, if $w \notin A$ then there is no proof string $p$ that can convince $V$ of $w$’s membership in $A$. Notice also that $V$’s running time on input $\langle w, p \rangle$ is independent of the length of $p$; it always runs in time $|w|^k$ no matter what $p$ is. Now, since in $|w|^k$ steps, $V$ cannot read more than $|w|^k - |w|$ bits of $p$, we can without loss of generality assume that $p$ is a string whose length
is bounded by a polynomial in the length of \( w \). Thus, we could informally say that \( A \) is polynomially verifiable, if for any string \( w \in A \) there is a “short” proof (of polynomial length) that can be efficiently checked (in polynomial time) by a verifier, and if \( w \notin A \) there is no proof that can convince a verifier.

A language being polynomially verifiable is equivalent to a problem having a nondeterministic polynomial time verifier.

**Theorem A.42** \( A \in \text{NP} \) if and only if \( A \) is polynomially verifiable.

**Proof** Consider \( A \in \text{NP} \), and let \( M \) be nondeterministic Turing machine recognizing \( A \) in time \( n^k \) for some \( k \). We can assume without loss of generality that \( M \) has at most two choices at any given step. The verifier \( V \) for \( A \) will work as follows. On input \( \langle w, p \rangle \), where \( p \) is a binary string, it will first copy \( w \) onto a work-tape, and compute \( n^k \). It will then simulate \( M \) for \( n^k \) steps using the work-tape with \( w \) as the input tape, taking \( p \) to be the sequence of nondeterministic choices. \( V \) accepts \( \langle w, p \rangle \) if \( M \) accepts \( w \) with \( p \) as the nondeterministic choices. Observe that \( V \) is a deterministic algorithm running in \( O(n^k) \) time on \( |w| = n \). Further \( A = \{ w | \exists p. V \text{ accepts } \langle w, p \rangle \} \).

Conversely, suppose \( V \) is a polynomial time verifier for \( A \). Suppose \( V \) runs in time \( |w|^k \) on input \( \langle w, p \rangle \). The nondeterministic algorithm \( M \) for \( A \) will work as follows. On input \( w \), \( M \) will guess a string \( p \) of length \( |w|^k \). Then \( M \) will simulate \( V \) on \( w \) and the guessed string \( p \), accepting if and only if \( V \) accepts. It is easy to see that \( L(M) = A \) and \( M \) runs in time \( O(n^k) \). \( \square \)

Thus, \( \text{NP} \) is the collection of all problems \( A \) whose membership question has short, efficiently checkable proofs. The question of whether all problems in \( \text{NP} \) have polynomial time algorithms — whether \( P \equiv \text{NP} \) — is thus the question of whether every problem that has a short, efficiently checkable proofs also have the property that these proofs can be found efficiently. Phrased in this manner, the likely answer seems to be no. There are also results that seem to suggest that \( P \) is likely to be not equal to \( \text{NP} \), though a firm resolution of this question has eluded researchers for the past 50 years.

### A.7.2 Reductions, Hardness and Completeness

In an effort to resolve the \( P \) versus \( \text{NP} \) question, researchers have tried to identify canonical problems whose study can help address this challenge. The goal is to identify, in some sense, the most difficult problems in \( \text{NP} \) such that either (a) they are candidate problems that may not have polynomial time algorithms, or (b) finding a polynomial time algorithms for these problems will constructively demonstrate that \( P = \text{NP} \). In order to identify such difficult problems, we need to be able to compare the difficulty of two problems. For this the most convenient technique is that of reductions. Unlike, many-one reductions introduced before, we will require that these be computed in polynomial time.
Definition A.43 A polynomial time reduction from $A$ to $B$ is a polynomial time computable function $f$ such that for every input $w$,

$$w \in A \text{ if and only if } f(w) \in B.$$ 

In such a case we say that $A$ is polynomial time reducible to $B$ and is denoted by $A \leq_P B$.

Example A.44 Consider the following problems.

$\text{SAT} = \{ \langle \varphi \rangle \mid \varphi \text{ is in CNF and } \varphi \text{ is satisfiable} \}$

$k$-$\text{COLOR} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph that can be colored using } k \text{ colors} \}$

Proposition 1.35 shows that for any graph $G$ and $k \in \mathbb{N}$ there is a set of clauses $\Gamma_{G,k}$ such that $G$ is $k$-colorable if and only if $\Gamma_{G,k}$ is satisfiable. The number of clauses in $\Gamma_{G,k}$ is proportional to the number of vertices and edges in $G$, and the each clauses has at most $k$-literals. It is also easy to see that $\Gamma_{G,k}$ can be constructed from $G$ in time that linear in the size of $G$. Thus, these observations together establish that $k$-$\text{COLOR} \leq_P \text{SAT}$.

Example A.45 A formula $\varphi$ in CNF is said to be in 3-CNF if every clause in $\varphi$ has exactly 3 literals. For example, $(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_4) \land (\neg x_2 \lor \neg x_3 \lor x_4)$ is not in 3-CNF, while $(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_2)$ is in 3-CNF. Recall the SAT problem is one whether given a formula $\varphi$ in CNF, we need to determine if $\varphi$ is satisfiable. A special case of this problem is one where the input formula is promised to be in 3-CNF. Formally we have,

$$3-$\text{SAT} = \{ \langle \varphi \rangle \mid \varphi \text{ is in 3-CNF and is satisfiable} \}.$$ 

Since 3-$\text{SAT}$ is a “special” version of SAT, the identity function is a reduction from 3-$\text{SAT}$ to SAT; thus, 3-$\text{SAT} \leq_P \text{SAT}$. It turns out the one also has a reduction the other way around.

The reduction from SAT to 3-$\text{SAT}$ is as follows. Consider a CNF formula $\varphi$; it will be convenient to this of $\varphi$ as a set of clauses. Our reduction will convert (in polynomial time) each clause $c \in \varphi$ into a 3-CNF formula $f(c)$ such that $c$ and $f(c)$ are satisfied by (almost) the same set of truth assignments. Then $f(\varphi) = \{ f(c) \mid c \in \varphi \}$, and it will be the case that $\varphi$ is satisfiable iff $f(\varphi)$ is satisfiable.

Let us now describe the translation of clauses. The translation of clause $c$ will depend on how many literals $c$ has. Let $c = \ell_1 \lor \ell_2 \lor \cdots \lor \ell_k$. Depending on $k$, we have the following cases.

Case $k = 1$  Let $u$ and $v$ be “new” propositions not used before. Define $f(c)$ to be

$$(\ell_1 \lor u \lor v) \land (\ell_1 \lor u \lor \neg v) \land (\ell_1 \lor \neg u \lor v) \land (\ell_1 \lor \neg u \lor \neg v).$$

Case $k = 2$  Let $u$ be a “new” proposition. $f(c)$ is given by

$$(\ell_1 \lor \ell_2 \lor u) \land (\ell_1 \lor \ell_2 \lor \neg u).$$
Case $k = 3$. In this case $f(c) = c$. 
Case $k > 3$. Let $y_1, y_2, \ldots, y_{k-3}$ be new propositions. Then $f(c)$ is

$$(\ell_1 \lor \ell_2 \lor y_1) \land (\ell_3 \lor \neg y_1 \lor y_2) \land (\ell_4 \lor \neg y_2 \lor y_3) \land \cdots \land (\ell_{k-2} \lor \neg y_{k-4} \lor y_{k-3}) \land (\ell_{k-1} \lor \ell_k \lor \neg y_{k-3})$$

It is easy to see that $f$ can be computed in time that is linear in the size of $\varphi$. Moreover, $\varphi$ is satisfiable iff $f(\varphi)$ is satisfiable (left as exercise). Thus, $f$ is a polynomial time reduction showing $\text{SAT} \leq_p 3\text{-SAT}$.

Proposition A.46  The following properties hold for polynomial time reductions.

- If $A \leq_P B$ then $\overline{A} \leq_P \overline{B}$.
- If $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.

Proof. Detailed proof of these observations is left as an exercise. But the sketch is as follows. If $f$ is a polynomial time reduction from $A$ to $B$ then $f$ is also a polynomial time reduction from $\overline{A}$ to $\overline{B}$. And if $f$ is a polynomial time reduction from $A$ to $B$ and $g$ is a polynomial time reduction from $B$ to $C$, then $g \circ f$ is a polynomial time reduction from $A$ to $C$.

Finally, polynomial time reductions do serve as a way to compare the computational difficulty of two problems. We show that if $A \leq_P B$ and $B$ is “easy” then $A$ is easy.

Theorem A.47  If $A \leq_P B$ and $B \in \mathsf{P}$ then $A \in \mathsf{P}$.

Proof. Let $f$ be a polynomial time reduction from $A$ to $B$ and let $M$ be a deterministic polynomial time algorithm recognizing $B$. Then the polynomial time algorithm $N$ for $A$ does the following: On input $w$, compute $f(w)$ and then run $M$ on $f(w)$. It is easy to see that $N$ recognizing $A$ from the properties of a reduction.

The tricky step is to argue that $N$ runs in polynomial time. Let us assume that $f$ is computed in time $n^k$ and let $M$ run in time $n^\ell$. Since $f$ can be computed in time $n^k$, it means that $|f(w)| \leq |w|^k$; this is because a single step in the computation of $f$ can produce at most one bit of $f(w)$. Therefore, the total running time of $N$ is $|w|^k$ (time to compute $f(w)$) + $(|w|^k)^\ell$ (time to run $M$ on $f(w)$ which is a string of length $|w|^k$). This is bounded by $O(n^{k\ell})$ which is polynomial.

In Theorem A.47, we could have replaced $\mathsf{P}$ by any of the complexity classes in Fig. A.3 that contain $\mathsf{P}$, and the proof would go through. Thus, polynomial time reductions are an appropriate lens by which measure the relative difficulty of problems that belong to complexity classes that contain $\mathsf{P}$.

Definition A.48 (Hardness and Completeness)

Let $C$ be a complexity class in Fig. A.3 that contains $\mathsf{P}$. $A$ is said to be $C$-hard iff for every $B \in C, B \leq_p A$.

$A$ is said to be $C$-complete iff $A \in C$ and $A$ is $C$-hard.
In other words, informally, a problem is $C$-hard if it is at least as difficult as any problem in $C$. It is $C$-complete if in addition it also belongs to $C$. Fixing $C$ to be $NP$, we could say that a problem is $NP$-complete if it is the “hardest” problem that belongs to $NP$. Because of their status as the most difficult problems in $NP$, they are candidate problems to study to help resolve the $P$ versus $NP$ question. This is captured by the following observation.

**Proposition A.49** If $A$ is $NP$-hard and $A \in P$ then $NP = P$.

**Proof** Consider any problem $B \in NP$. Since $B \leq_P A$ and $A \in polytime$, by Theorem A.47, we have $B \in P$. \qed

In the absence of a firm resolution of the $P$ versus $NP$ questions, classifying a problem as $NP$-hard suggests that it is unlikely that the problem has a polynomial time algorithm given our belief that $P \neq NP$.

Many natural problems are $NP$-complete. The historically (and pedagogically) first problem known to be $NP$-complete is SAT (Cook-Levin Theorem Theorem 1.23). Observe that from Fig. A.3, we have $P \subseteq NP$ and $P \subseteq \text{coNP}$. From this we can conclude that $P \subseteq NP \cap \text{coNP}$. Related but independent of the $P$ versus $NP$ question is whether $P = NP \cap \text{coNP}$. This question also remains open. Many problems that were previously known to be in $NP \cap \text{coNP}$ were proved to be in $P$ years later. Two classical examples are Linear programming that was shown to be in $P$ by Khachiyan in 1979 and testing whether a number is prime, which was proved by Agarwal-Kayal-Saxena in 2002 to be in $P$. However, there are some natural problems in $NP \cap \text{coNP}$ whose status with respect to $P$ is still unresolved. One is the problem of solving parity games, and the other is the factoring problem.