# 4 Complexity of Logic

## 4.1 Introduction to logic

We start the section with an introduction to logic. A *Boolean variable* is a variable that can take the two truth values *true* and *false*. Boolean variables can be combined to *Boolean expressions* using the Boolean connectives $\vee$ (logical "or"), $\wedge$ (logical "and"), and $\neg$ (logical "not"). Formally, a Boolean expression can be any one of

(a) a Boolean variable,
(b) an expression of the form $\neg\phi_1$, where $\phi_1$ is a Boolean expression,
(c) an expression of the form $(\phi_1 \vee \phi_2)$, where $\phi_1$ and $\phi_2$ are Boolean expressions, or
(d) an expression of the form $(\phi_1 \wedge \phi_2)$, where $\phi_1$ and $\phi_2$ are Boolean expressions.

In case (b) the expression is called the *negation* of $\phi_1$; in case (c), it is the *disjunction* of $\phi_1$ and $\phi_2$; and in case (d), it is the *conjunction* of $\phi_1$ and $\phi_2$. An expression of the form $x_i$ or $\neg x_i$, where $x_i$ is a Boolean variable, is called a *literal*.

We shall find it convenient to use two more Boolean connectives: $(\phi_1 \Rightarrow \phi_2)$ is a shorthand for $(\neg\phi_1 \vee \phi_2)$, and $(\phi_1 \Leftrightarrow \phi_2)$ is a shorthand for $((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1))$.

The definitions above only specify the *syntax* of Boolean expressions. What gives a logical expression life is its *semantics*, or meaning. The semantics or Boolean expressions is relatively simple: it can be true or false depending on the truth values of the Boolean variables involved in it.

A *truth assignment* $T$ is a mapping from a finite set $X$ of Boolean variables to the set of truth values $\{true, false\}$. Let $\phi$ be a Boolean expression. The truth assignment *approriate to* $\phi$ is defined as the assignment $T_\phi$ that contains all Boolean variables in $\phi$. We define inductively what it means for $T$ to satisfy $\phi$, written $T \models \phi$ ($\phi_1$ and $\phi_2$ denote Boolean expressions):

- $\phi = x_i$ for some Boolean variable $x_i$: $T \models \phi$ if $T(x_i) = \mathit{true}$
- $\phi = \neg\phi_1$: $T \models \phi$ if $T \not\models \phi_1$
- $\phi = (\phi_1 \vee \phi_2)$: $T \models \phi$ if $T \models \phi_1$ or $T \models \phi_2$
- $\phi = (\phi_1 \wedge \phi_2)$: $T \models \phi$ if $T \models \phi_1$ and $T \models \phi_2$

Consider, for example, the Boolean expression $\phi = ((\neg x_1 \vee x_2) \wedge x_3)$. An example of an appropriate truth assignment is one that has $T(x_1) = T(x_3) = \mathit{true}$ and $T(x_2) = \mathit{false}$. Using the inductive definition of an assignment satisfying $\phi$, it is easy to check that $T \not\models \phi$.

We say that two expressions $\phi_1, \phi_2$ are *equivalent*, written $\phi_1 \equiv \phi_2$, if for any truth assignment $T$ appropriate to both of them, $T \models \phi_1$ if and only if $T \models \phi_2$. If two Boolean expressions are equivalent, they can be considered as different representations of one and the same object, and can be used interchangeably. Next we summarize some useful equivalences. (1) and (2) are commutativity properties, (4) and (5) are associativity properties, and (8) and (9) are the De Morgan's laws.

**Proposition 4.1** *Let $\phi_1$, $\phi_2$, and $\phi_3$ be arbitrary Boolean expressions. Then*

1. $(\phi_1 \vee \phi_2) \equiv (\phi_2 \vee \phi_1)$
2. $(\phi_1 \wedge \phi_2) \equiv (\phi_2 \wedge \phi_1)$

*3.* $\neg\neg\phi_1 \equiv \phi_1$
*4.* $((\phi_1 \vee \phi_2) \vee \phi_3) \equiv (\phi_1 \vee (\phi_2 \vee \phi_3))$
*5.* $((\phi_1 \wedge \phi_2) \wedge \phi_3) \equiv (\phi_1 \wedge (\phi_2 \wedge \phi_3))$
*6.* $((\phi_1 \wedge \phi_2) \vee \phi_3) \equiv ((\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3))$
*7.* $((\phi_1 \vee \phi_2) \wedge \phi_3) \equiv ((\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3))$
*8.* $\neg(\phi_1 \vee \phi_2) \equiv (\neg\phi_1 \wedge \neg\phi_2)$
*9.* $\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_2)$
*10.* $\phi_1 \vee \phi_1 \equiv \phi_1$
*11.* $\phi_1 \wedge \phi_1 \equiv \phi_1$

The equivalences can simply be shown by the *truth table method* (i.e., listing and comparing all possible truth assignments). Proposition 4.1 allows us to simplify the notation of Boolean expressions. For instance, instead of $(((x_1 \vee \neg x_3) \vee x_2) \vee x_4 \vee (x_2 \vee x_5))$ we can write $(x_1 \vee x_2 \vee \neg x_3 \vee x_4 \vee x_5)$. As a shorthand for long disjunctions and long conjunctions, we will use $\bigwedge_{i=1}^n \phi_i$ instead of $(\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n)$ and $\bigvee_{i=1}^n \phi_i$ instead of $(\phi_1 \vee \phi_2 \vee \ldots \vee \phi_n)$.

A Boolean expression $\phi$ is in *conjunctive normal form* (or CNF) if $\phi = \bigwedge_{i=1}^n C_i$, where each of the $C_i$s is the disjunction of one or more literals. The $C_i$s are called the *clauses* of the expression. Similarly, we say that an expression $\phi$ is in *disjunctive normal form* (or DNF) if $\phi = \bigvee_{i=1}^n D_i$, where each of the $D_i$s is the conjunction of one or more literals. The $D_i$s are called the *implicants* of the expression.

**Proposition 4.2** *Every Boolean expression can be transformed into an expression in conjunctive normal form, and to one in disjunctive normal form.*

**Proof.** For a proof see, for instance, p. 76 of "Computational Complexity". □

One has to be careful with this proposition, because there are classes of Boolean expressions that have exponentially longer CNFs or DNFs.

We say that a Boolean expression $\phi$ is *satisfiable* if there is a truth assignment $T$ appropriate to it such that $T \models \phi$. $\phi$ is called *valid* or a *tautology* if $T \models \phi$ for all $T$ appropriate to $\phi$. Thus, a valid expression must be satisfiable. If $\phi$ is valid, we simply write $\models \phi$.

In order to process a boolean expression by an algorithm, it can be represented as a string over an alphabet that contains the symbols $x$, 0, and 1 (useful for writing variable names with indices in binary), and also the symbols (, ), $\vee$, $\neg$, and $\wedge$ required by the syntax of Boolean expressions. The *length* of a Boolean expression is the length of the corresponding string. The most important decision problem concerning Boolean expressions is the following:

$$SAT = \{\langle\phi\rangle : \ \phi \text{ is a satisfiable Boolean expression in conjunctive normal form}\} .$$

We will see how difficult it is to solve this problem.

An extension of the Boolean expressions are the *quantified Boolean expressions*. In addition to the Boolean expressions, quantified Boolean expressions use *existential quantifiers* (the logical "there exists"), written $\exists$, and *universal quantifiers* (the logical "for all"), written $\forall$. Formally, a quantified Boolean expression can be any one of

(a) a Boolean expression,

(b) an expression of the form $\exists x\,\phi$, where $\phi$ is a quantified Boolean expression,
(c) an expression of the form $\forall x\,\phi$, where $\phi$ is a quantified Boolean expression, or
(d) an expression of the form $\neg\phi_1$, $\phi_1 \vee \phi_2$, or $\phi_1 \wedge \phi_2$, where $\phi_1$ and $\phi_2$ are quantified Boolean expressions.

An appearance of a variable in the text of an expression $\phi$ that does not directly follow a quantifier is called an *occurance* of $x$ in $\phi$. Occurances can be *free* or *bound*. If $\forall x\,\phi$ is an expression, any occurance of $x$ in $\phi$ is called bound. It is also bound when occuring in any expression that contains $\forall x\,\phi$ as a subexpression. All other occurances are called free. A variable that has a free occurance in $\phi$ is called a *free variable* of $\phi$. A *sentence* is an expression without free variables. Hence, an example of a correct quantified Boolean expression would be $(\forall x(x \wedge \exists y(y \vee z))) \vee (x \vee \neg y)$.

As for Boolean expressions, a truth assignment appropriate to a quantified Boolean expression $\phi$ is defined as the assignment $T_\phi$ that contains all Boolean variables in $\phi$. We define inductively what it means for $T$ to satisfy $\phi$ ($\phi_1$ and $\phi_2$ will denote quantified Boolean formulas):

- $\phi = x_i$ for some Boolean variable $x_i$: $T \models \phi$ if $T(x_i) = \mathit{true}$
- $\phi = \neg\phi_1$: $T \models \phi$ if $T \not\models \phi_1$
- $\phi = (\phi_1 \vee \phi_2)$: $T \models \phi$ if $T \models \phi_1$ or $T \models \phi_2$
- $\phi = (\phi_1 \wedge \phi_2)$: $T \models \phi$ if $T \models \phi_1$ and $T \models \phi_2$
- $\phi = \exists x\,\phi_1$: $T \models \phi$ if $T_{x=true} \models \phi_1$ or $T_{x=false} \models \phi_1$
- $\phi = \forall x\,\phi_1$: $T \models \phi$ if $T_{x=true} \models \phi_1$ and $T_{x=false} \models \phi_1$

We say that two quantified Boolean expressions are *equivalent*, written $\phi_1 \equiv \phi_2$, if for any truth assignment $T$ appropriate to both of them, $T \models \phi_1$ if and only if $T \models \phi_2$. Next we list some useful equivalences.

**Proposition 4.3** *Let $\phi_1$ and $\phi_2$ be arbitrary quantified Boolean expressions. Then, in addition to the equivalences in Proposition 4.1,*

1. $\neg\exists x\,\phi_1 \equiv \forall x\,\neg\phi_1$
2. $\neg\forall x\,\phi_1 \equiv \exists x\,\neg\phi_1$
3. $\exists x(\phi_1 \vee \phi_2) \equiv (\exists x\,\phi_1 \vee \exists x\,\phi_2)$
4. $\forall x(\phi_1 \wedge \phi_2) \equiv (\forall x\,\phi_1 \wedge \forall x\,\phi_2)$
5. *if $x$ does not appear free in $\phi_2$, $\forall x(\phi_1 \wedge \phi_2) \equiv (\forall x\,\phi \wedge \phi_2)$ (the same holds for $\exists$)*
6. *if $x$ does not appear free in $\phi_2$, $\forall x(\phi_1 \vee \phi_2) \equiv (\forall x\,\phi \vee \phi_2)$ (the same holds for $\exists$)*
7. *if $y$ does not appear in $\phi$, $\forall x\,\phi \equiv \forall y\,\phi[x \leftarrow y]$ (the same holds for $\exists$)*
   *($\phi[x \leftarrow y]$ means $\phi$ with $x$ replaced by $y$)*

*It does* not *hold in general that*

- $\forall x_1\,\exists x_2\,\phi \equiv \exists x_2\,\forall x_1\,\phi$
- $\exists x(\phi_1 \wedge \phi_2) \equiv (\exists x\,\phi_1 \wedge \exists x\,\phi_2)$
- $\forall x(\phi_1 \vee \phi_2) \equiv (\forall x\,\phi_1 \vee \forall x\,\phi_2)$

The proposition allows us to simplify the notation of quantified Boolean expressions. A quantified Boolean expression is in *prenex normal form* if it consists of a sequence of quantifiers, followed by an expression that is free of quantifiers. For instance, the prenex normal form of

$$(\forall x_1(x_1 \wedge (\forall x_2(x_2 \vee x_3)))) \wedge (\forall x_3(\exists x_4(x_3 \wedge x_4)))$$

is

$$\forall x_1 \forall x_2 \forall x_3 \exists x_4 (x_1 \wedge (x_2 \vee x_5)) \wedge (x_3 \wedge x_4)$$

The next proposition states that this can be done for every quantified Boolean expression.

**Proposition 4.4** *Every quantified Boolean expression can be transformed into a quantified Boolean expression in prenex normal form.*

Unlike Proposition 4.2, the length of the expression in prenex normal form is always polynomial in the length of the original expression. Combining Proposition 4.2 with Proposition 4.4, we obtain:

**Proposition 4.5** *Every quantified Boolean expression can be transformed into an expression in conjunctive prenex normal form.*

A quantified Boolean expression $\phi$ is *satisfiable* if there is a truth assignment $T$ appropriate to it such that $T \models \phi$. $\phi$ is called *valid* if $T \models \phi$ for all $T$ appropriate to $\phi$. The most important decision problem concerning quantified Boolean expressions is the following:

$$\begin{aligned} QSAT \;\; = \;\; & \{\langle\psi\rangle : \psi \text{ is a valid quantified Boolean formula of the form } \exists x_1 \forall x_2 \exists x_3 \ldots Q_m x_m \phi \\ & \text{for some } m \in \mathbb{N}, \text{ where } \phi \text{ is an expression in CNF with variables } x_1, \ldots, x_m\} \;. \end{aligned}$$

## 4.2 Reductions and Completeness

Let $L_1, L_2 \in \Sigma^*$ be any two languages. $L_1$ is called *reducible* to $L_2$, written $L_1 \leq L_2$, if there is a computable function $f : \Sigma^* \to \Sigma^*$ with the property that $x \in L_1 \;\Leftrightarrow\; f(x) \in L_2$. $L_1$ and $L_2$ are called *equivalent*, written $L_1 \equiv L_2$, if $L_1 \leq L_2$ and $L_2 \leq L_1$. With these definitions, $\leq$ forms an order and $\equiv$ forms an equivalence relation. The definition of $\leq$ has the following important consequence.

**Proposition 4.6** *Let $L_1$ and $L_2$ be any two languages. If $L_2$ is recursive and $L_1 \leq L_2$, then also $L_1$ is recursive.*

**Proof.** If $L_1 \leq L_2$, then there is a computable function $f$ with $x \in L_1 \;\Leftrightarrow\; f(x) \in L_2$. Since $f$ is computable, there must be a Turing machine $M_f$ that converts any input $x$ into $f(x)$ in finite time. Let $M_2$ be the Turing machine that decides $L_2$. Using $M_f$ and $M_2$ we can build a Turing machine $M_1$ that decides $L_1$ in the following way:

- start $M_f$ on input $x$ to compute $f(x)$
- start $M_2$ on input $f(x)$
- accept if and only if $M_2$ accepts

Since $M_1$ accepts input $x$ if and only if $M_2$ accepts input $f(x)$, $M_1$ accepts exactly those inputs that are in $L_1$. Furthermore, $M_1$ always halts. Hence, $L_1$ is recursive. □

A similar result can be shown for recursively enumerable languages. Another important consequence is:

**Proposition 4.7** *Let $L_1$ and $L_2$ be any two languages. If $L_1$ is not recursive and $L_1 \leq L_2$, then also $L_2$ cannot be recursive.*

**Proof.** Suppose that $L_2$ is recursive. Then, by the construction for $M_1$ above, we could construct a Turing machine that decides $L_1$. This, however, contradicts our assumption that $L_1$ is not recursive, which proves the proposition. □

Also here a similar result can be shown for recursively enumerable languages. Thus, the reduction principle gives us a way of proving the hardness of solving problems with the help of other problems and grouping problems into classes of the same hardness.

Also, reductions with time and/or space restrictions have been considered. A language $L_1$ is called *polynomially reducible* to $L_2$, written $L_1 \leq_p L_2$ if there is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ with the property that $x \in L_1 \Leftrightarrow f(x) \in L_2$. Consider any complexity class $C$. A language $L$ is called $C$*-hard* if for all $L' \in C$, $L' \leq_p L$. If in addition $L \in C$, then $L$ is called $C$*-complete*. This has the following important consequences.

- If $\mathrm{P} \subset C$ then for every $C$-hard language $L$, $L \notin \mathrm{P}$.
- If $L \in \mathrm{P}$ for some $C$-complete language $L$, then $C \subseteq \mathrm{P}$.

Thus, for example, if an NP-complete problem were in P, then P = NP.

## 4.3 Complexity of SAT and QSAT

**Theorem 4.8 (Cook/Levin)** *$SAT$ is* NP*-complete.*

**Proof.** In order to prove the theorem, we have to show that

1. $SAT \in \mathrm{NP}$ and
2. $L \leq_p SAT$ for all $L \in \mathrm{NP}$.

Proof of (1): Given a satisfiable Boolean expression, a nondeterministic Turing machine can "guess" the satisfying truth assignment and verify it in polynomial time.

Proof of (2): Consider an arbitrary $L \in \mathrm{NP}$. Then there is a nondeterministic single-tape Turing machine $M$ that decides $L$ in polynomial time. More precisely, there are constants $c, k \in \mathbb{N}$ so that the runtime of $M$ when starting on an input of length $n$ is at most $c \cdot n^k$ for all $n \in \mathbb{N}$. In order to obtain a polynomial time computable function $f$ with $x \in L \Leftrightarrow f(x) \in SAT$, we use the following strategy:

Given some input $x$, $|x| = n$, let $T = c \cdot n^k$. Consider the set of Boolean variables

$$V_t = \{d_{i,c,t} : \ c \in \Gamma, -T \leq i \leq T\} \cup \{h_{i,t} : \ -T \leq i \leq T\} \cup \{s_{q,t} : q \in Q\} \ ,$$

where $t \in \{1, \ldots, T\}$. They have the following meaning:

- $d_{i,c,t}$: the contents of cell $i$ at time $t$ is $c$
- $h_{i,t}$: the position of the head of $M$ at time $t$ is cell $i$

- $s_{q,t}$: the state of $M$ at time $t$ is equal to $q$

Since $M$ started on $x$ never leaves the tape area from cell $-T$ to cell $T$, we can restrict ourselves to consider only these tape cells to represent a configuration as a Boolean expression. $V_t$ is used to encode the $t$th configuration $C_t$. Any such configuration has to fulfill:

- cell $i$ has a unique contents:

$$\left(\bigvee_{c\in\Gamma} d_{i,c,t}\right) \wedge \neg\left(\bigvee_{\{c_1,c_2\}\subseteq\Gamma} (d_{i,c_1,t} \wedge d_{i,c_2,t})\right) = \left(\bigvee_{c\in\Gamma} d_{i,c,t}\right) \wedge \bigwedge_{\{c_1,c_2\}\subseteq\Gamma} (\neg d_{i,c_1,t} \vee \neg d_{i,c_2,t})$$

- the head of $M$ has a unique position:

$$\left(\bigvee_{i\in\{-T,\ldots,T\}} h_{i,t}\right) \wedge \neg\left(\bigvee_{\{i_1,i_2\}\subseteq\{-T,\ldots,T\}} (h_{i_1,t} \wedge h_{i_2,t})\right)$$

- $M$ has a unique state:

$$\left(\bigvee_{q\in Q} s_{q,t}\right) \wedge \neg\left(\bigvee_{\{q_1,q_2\}\subseteq Q} (s_{q_1,t} \wedge s_{q_2,t})\right)$$

To ensure that we have an accepting computational path $C_1 \to C_2 \to \ldots \to C_T$, the following additional conditions have to hold:

- $C_1$ is the initial configuration with input $x = x_0 \ldots x_{n-1}$:

$$s_{q_0,1} \wedge h_{0,1} \wedge \left(\bigwedge_{i=0}^{n-1} d_{i,x_i,1}\right) \wedge \left(\bigwedge_{i\in\{-T,\ldots,T\}-\{1,\ldots,n\}} d_{i,B,1}\right)$$

- $C_T$ is a final configuration:

$$\bigvee_{q\in F} s_{q,T}$$

- for all $t \in \{1, \ldots, T\}$, $C_t \to C_{t+1}$ or $C_t = C_{t+1}$: to express this as a Boolean expression will be an assignment.

The conjunction of all Boolean expressions above results in a Boolean expression $\phi(x)$ in CNF that is satisfiable if and only if there is an accepting computation of $M$ with at most $T$ steps. It is easy to see that the size of $\phi(x)$ is polynomial in $n$ and that $\phi(x)$ can be computed in time polynomial in $n$. Hence, the function $f$ with $f(x) = \phi(x)$ is polynomial time computable, and therefore $L \leq_p SAT$. □

**Theorem 4.9 (Stockmeyer and Meyer)** *$QSAT$ is* PSPACE*-complete.*

**Proof.** In order to prove the theorem, it suffices to show that

1. $QSAT \in$ PSPACE and
2. $L \leq_p QSAT$ for all $L \in$ PSPACE.

Proof sketch of (1): The rules given above for determining the truth value of an expression $\psi = \exists x_1 \forall x_2 \exists x_3 \ldots Q_m x_m \, \phi$ can be seen as a complete binary tree of depth $m+1$ in which level $i \in \{1, \ldots, m\}$ represents the evaluation of quantifier $i$ and level $m+1$ represents the evaluation of $\phi$ for a given truth assignment of the Boolean variables (see Figure 1). Evaluating this tree in a depth-first-search manner requires only polynomial space.

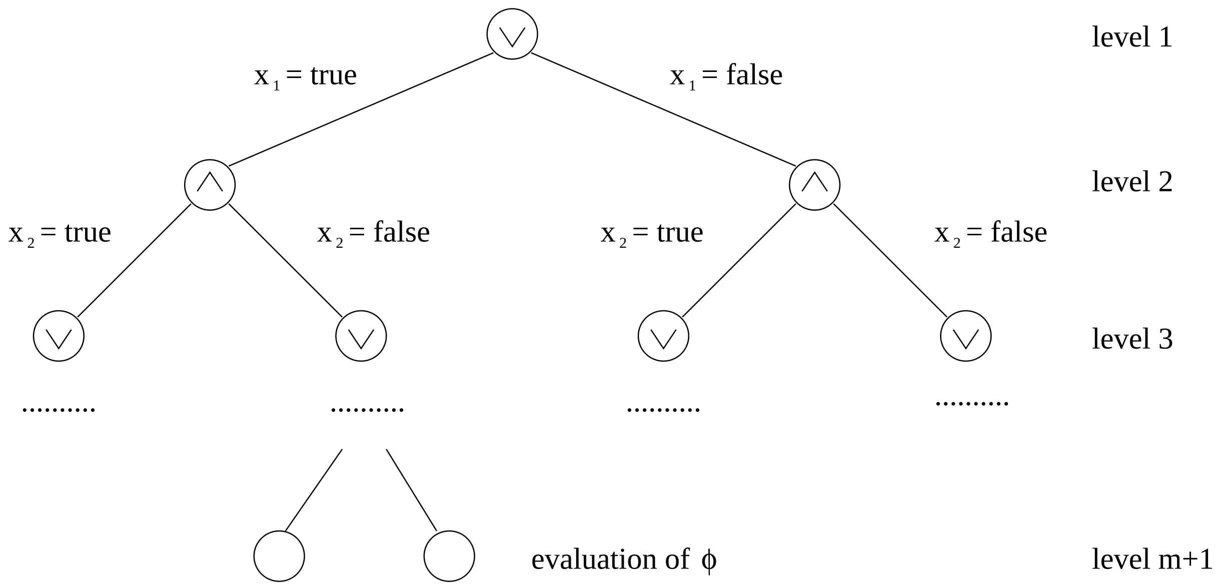


Figure 1: The structure of an evaluation tree for $\psi$.

Proof sketch of (2): Consider an arbitrary $L \in$ PSPACE. Then there is a deterministic Turing single-tape machine $M$ that decides $L$ in polynomial space. Or more precisely, there is a polynomial $s(n)$ such that $M$ decides any input of length $n$ with at most $s(n)$ tape cells. Since $M$ decides $L$ and only uses $s(n)$ space this, we know from Section 3 that $M$ needs at most $T = |Q| \cdot s(n) \cdot |\Gamma|^{s(n)}$ many steps. As in the proof by Cook, we can construct a Boolean expression in CNF that is true if and only if the truth assignment of the Boolean variables represents a valid configuration. Furthermore, we can give Boolean expressions $E(V_t, V_{t+1})$ that are true if and only if the truth assignments for $V_t$ and $V_{t+1}$ represent configurations $C_t$ and $C_{t+1}$ and either $C_t = C_{t+1}$ or $C_t \to C_{t+1}$. Hence,

$$x \in L \;\Leftrightarrow\; S(V_1) \wedge \left( \bigwedge_{t=1}^{T-1} E(V_t, V_{t+1}) \right) \wedge F(V_T) \,,$$

where $S(V_1)$ is a Boolean expression for "$V_1$ represents the initial configuration" and $F(V_T)$ is a Boolean expression for "$V_T$ represents the final configuration". The problem is that the length of this expression is far too big: it is $\Omega(2^{s(n)})$.

To remove this problem, we use the recursive approach of Savitch. Let "$V_1 \overset{\leq 2^r}{\to} V_2$" be the statement "the configuration represented by $V_2$ can be reached from the configuration represented by $V_1$ in at most $2^r$ steps. For $r = 0$ we can directly represent it as a Boolean expression. For $r > 0$ it holds:

$$(V_1 \overset{\leq 2^r}{\to} V_2) \;\equiv\; \exists V_3 \left( Config(V_3) \wedge (V_1 \overset{\leq 2^{r-1}}{\longrightarrow} V_3) \wedge (V_3 \overset{\leq 2^{r-1}}{\longrightarrow} V_2) \right) ,$$

where the Boolean expression $Config(V_3)$ checks whether $V_3$ represents a configuration. To "compress" this formula in the fashion of Savitch, we use the following trick:

$$\begin{aligned} (V_1 \overset{\leq 2^r}{\to} V_2) \;\equiv\; & \exists V_3 \forall V' \forall V''\, Config(V_3) \wedge \\ & \left( (((V' = V_1) \wedge (V'' = V_3)) \vee ((V' = V_3) \wedge (V'' = V_2))) \Rightarrow (V' \overset{\leq 2^{r-1}}{\longrightarrow} V'') \right) . \end{aligned}$$

It is not difficult to see that with this trick we can express $\bigwedge_{t=1}^{T-1} E(V_t, V_{t+1})$ by a quantified Boolean expression in conjunctive prenex normal form of size $O(s(n)^2 \cdot \log T) = O(s(n)^3)$. This expression is true if and only if there is an accepting computation of $M$ with at most $T$ steps. Since the transformation is polynomial time computable, $L \leq_p QSAT$. □

## 4.4 References


- S.A. Cook. The complexity of theorem-proving procedures. *Proc. of the 3rd IEEE Symp. on Foundations of Computer Science*, pp. 151–158, 1971.
- L.A. Levin. Universal sorting problems. *Problems of Information Transmission* 9, pp. 265–266, 1973.
- C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, Reading, 1994.
- L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. *Proc. of the 5th ACM Symp. on Theory of Computing*, pp. 1–9, 1973.