

# Introduction to the Theory of Complexity Classes and Logic

Stefan Kunkel

June 5, 2009

## **Abstract**

This paper — written for the *Joint Advanced Student School 2009* in St. Petersburg — gives an introduction to the theory of complexity classes and logic for the and sets a baseline of knowledge and notation about these topics. It assumes that the attendees are already familiar with them and as such skips lengthy explanations and sketches or skips many of the standard proofs.

The first part covers Turing-machines including determinism and non-determinism, input and output, space and time complexity, some complexity classes and completeness. It mostly follows Papadimitriou.

The second shorter part deals with the very basics of propositional logic and proof systems and gives a short example of one. Most of it is adapted from Buss and Pudlak.

# Contents

<b>I</b>	<b>Complexity Theory</b>	<b>2</b>
<b>1</b>	<b>Languages and Turing-machines</b>	<b>2</b>
1.1	Turing-machines . . . . .	2
1.2	Nondeterminism . . . . .	6
<b>2</b>	<b>Complexity Classes</b>	<b>7</b>
2.1	$P$ and $NP$ . . . . .	7
2.2	Space Complexity . . . . .	9
2.3	Relation between various Complexity Classes . . . . .	11
2.4	Completeness . . . . .	12
<b>II</b>	<b>Propositional Logic</b>	<b>14</b>
<b>3</b>	<b>Abstract Proof Systems</b>	<b>16</b>
<b>4</b>	<b>Example of a Propositional Proof System: Resolution</b>	<b>17</b>

# Part I

## Complexity Theory

### 1 Languages and Turing-machines

For computation and its input and output to make sense we first need a way to describe a “language” formally to express ourselves.

#### DEFINITION 1                      **Language**

- $\Sigma$  denotes a finite, non-empty set called alphabet; its elements are called symbol
- For symbols  $l_1, l_2, \dots, l_m \in \Sigma$  we write  $l_1 l_2 \dots l_m$  for its concatenation; concatenated symbols are called words
- The set of all words constructed by concatenating symbols in  $\Gamma \subseteq \Sigma$  is denoted by  $\Gamma^*$
- A subset  $L \subset \Sigma^*$  is called language
- For a word  $w$  we write  $|w|$  for its length.
- Shorthand notation:
  - The concatenation of words  $w_1 = l_1 \dots l_m$ ,  $w_2 = k_1 \dots k_n$  is defined as  $w_1 w_2 := l_1 \dots l_m k_1 \dots k_n$ .
  - Let  $\Gamma$  be a set of words. Then  $\Gamma^*$  denotes the set of all words constructed by concatenating the words in  $\Gamma$ .
  - For a letter  $a$  we write  $a^*$  instead of  $\{a\}^*$ ; similarly we write  $w^*$  instead of  $\{w\}^*$  for a word  $w$ .

Examples are well known: The Latin alphabet is an alphabet in our sense and English is an example of a language in our sense.

#### 1.1 Turing-machines

Now that we have a way of expressing ourselves we can formalize “computation”.

#### DEFINITION 2                      **Turing-machine**

A (deterministic) Turing-machine  $M$  is a quadruple  $M = (Q, \Sigma, \delta, s)$ , where  $Q$  is a finite set of states and  $\Sigma$  is an alphabet containing the special symbols  $\sqcup$ , the blank symbol, and  $\triangleright$ , the first symbol;  $s \in Q$  is the initial state.  $Q$  and  $\Sigma$  are disjoint.

Finally  $\delta : Q \times \Sigma \rightarrow (Q \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, \text{—}\}$  is called transition function. We assume that the halting state  $h$ , the accepting state “yes”, the rejecting state “no” and the cursor direction  $\rightarrow$ ,  $\leftarrow$  and  $\text{—}$  are neither in  $Q$  nor in  $\Sigma$ .

A Turing-machine can be seen as a cursor on a string of symbols and having a definite internal state. Every time the cursor moves over the end of the string a new  $\sqcup$  is inserted on the string.

The function  $\delta$  is the “program” of the machine. It specifies, for each combination of the current state  $q \in Q$  and current symbol  $a \in \Sigma$ , a triple  $\delta(q, l) = (p, b, D)$ .  $p$  is the next state,  $b$  is the symbol, which is written over  $a$ , and  $D \in \{\rightarrow, \leftarrow, \text{—}\}$  is the direction in which the cursor will move.

By definition a Turing-machine always starts on the symbol  $\triangleright$ . It cannot overwrite  $\triangleright$  and will always upon reading  $\triangleright$  move to the right.

Now define a way of asking Turing-machines and give them a way to answer us.

### DEFINITION 3 Input and Output of a Turing-machine

A Turing-machine  $M$  may be initialized with a word  $x$  written on its string. This word  $x$  is called input of  $M$  and for the resulting Turing-machine we write  $M(x)$ .

If a Turing-machine  $M$  halts on input  $x$  — i.e.  $M$  is in one of its three halting states  $h$ , “yes” or “no” — we can define its output: If  $M$  is in state “yes” it is said to accept  $x$  and  $M(x) = \text{"yes"}$ ; if  $M$  is in state “no” it is said to reject  $x$  and  $M(x) = \text{"no"}$ . Otherwise it is in state  $h$  and the output of  $M$  is the word on the string of  $M$ : by definition this string starts with a  $\triangleleft$ , followed by a word  $y$ , whose last symbol is not an  $\sqcup$ , possibly followed by a number of  $\sqcup$ -symbols. Then we consider  $y$  to be the output of  $M$  and we write  $M(x) = y$ .

If  $M$  does not halt on input  $x$ , we write  $M(x) = \nearrow$ .

Let us see a first and easy example of a Turing-machine in action:

**Example 1**  $M = (Q, \Sigma, \delta, s)$ , where  $Q = \{s\}$ ,  $\Sigma = \{0, 1, \sqcup, \triangleright\}$  and  $\delta$  as in the table below. We examine the behavior of  $M$  with the input 101:

$q \in Q$	$l \in \Sigma$	$\delta(q, l)$	
$s$	$\triangleright$	$(s, \triangleright, \rightarrow)$	0. $s, \triangleright 101$
$s$	0	$(s, 1, \rightarrow)$	1. $s, \triangleright \underline{1}01$
$s$	1	$(s, 0, \rightarrow)$	2. $s, \triangleright 0\underline{0}1$
$s$	$\sqcup$	$(h, \sqcup, \text{—})$	3. $s, \triangleright 01\underline{1}$
			4. $s, \triangleright 010\underline{\sqcup}$
			5. $h, \triangleright 010\underline{\sqcup}$

So the output is  $M(101) = 010$ .

To reason about Turing-machines we need a better way to talk about the computation going on and therefore we somehow need to talk about the current “overall state” of the machine and the transition to the next one respectively all states adopted.

**DEFINITION 4 Configuration**

A configuration of a Turing-machine  $M = (Q, \Sigma, \delta, s)$  contains all information of the current state of  $M$ . Formally it is a triple  $(q, w, u)$ , where  $q \in Q$  is the current state of  $M$ ,  $w$  is the word on the left of the cursor including the symbol scanned by the cursor and  $u$  is the word on the right.

This is the “overall state” a machine is currently in. In the example  $M$  is at step 3. in the configuration  $(s, 011, \epsilon)$ .

**DEFINITION 5 Yields in one step**

Let  $M$  be a Turing-machine. We say that a configuration  $(q, w, u)$  yields the configuration  $(q', w', u')$  in one step, denoted by  $(q, w, u) \xrightarrow{M} (q', w', u')$ , if a step of the machine from configuration  $(q, w, u)$  results in the configuration  $(q', w', u')$ .

Formally, it means that the following holds: Let  $l$  be the last symbol of  $w$  and  $\delta(q, l) = (q', l', D)$ . If  $D = \Rightarrow$ , then  $w'$  is  $w$  with  $l$  replaced by  $l'$  and appended by the first symbol of  $u$ ;  $u'$  is  $u$  with its first symbol removed. If  $D = \Leftarrow$ , then  $w'$  is  $w$  with  $l$  omitted from its end and  $u'$  is  $l'u$ . Finally if  $D = -$ , then  $w'$  is  $w$  with  $l$  replaced by  $l$  and  $u' = u$ .

This is the transition between states.

**DEFINITION 6 Yields**

Now that we have defined yields in one step, we define yields to be its transitive closure: That is, we say a configuration  $(q, w, u)$  yields  $(q'', w'', u'')$  in  $k$  steps, written as  $(q, w, u) \xrightarrow{M^k} (q'', w'', u'')$ , if there is a configuration  $(q', w', u')$  such that  $(q, w, u) \xrightarrow{M^{k-1}} (q', w', u')$  and  $(q', w', u') \xrightarrow{M} (q'', w'', u'')$ . Finally, we say a configuration  $(q, w, u)$  yields a configuration  $(q', w', u')$ , if there is a  $k \geq 0$  such that  $(q, w, u) \xrightarrow{M^k} (q', w', u')$ . In this case we write  $(q, w, u) \xrightarrow{M^*} (q', w', u')$ .

Finally this formalizes the notion of all the states adopted by the machine. In the example  $(s, \triangleright, 101)$  yields  $(h, \triangleright 010, \sqcup)$ .

With these concepts made clear we can describe the behavior of Turing-machines on whole sets of words using the halting states “yes” and “no” and define computation of a function using “ $h$ ”.

**DEFINITION 7 Deciding and accepting languages**

Let  $L \subset (\Sigma \setminus \{\sqcup\})^*$  be a language and  $M$  be a Turing-machine. If the output of  $M$  for any word  $x$  is either “yes”, when  $x \in L$ , or “no”, when  $x \notin L$  then we say, that  $M$  decides  $L$  and  $L$  is called a recursive language.

If  $M(x) = \text{“yes”}$ , when  $x \in L$ , and  $M(x) = \text{“}\surd\text{”}$ , when  $x \notin L$ , then we say  $M$  simply accepts  $L$  and  $L$  is called recursively enumerable.

**DEFINITION 8 Computation of functions**

Let  $f : (\Sigma \setminus \{\sqcup\})^* \rightarrow \Sigma^*$  be function and  $M$  be a Turing-machine with alphabet  $\Sigma$ . We say that  $M$  computes  $f$  if, for any word  $x \in (\Sigma \setminus \{\sqcup\})^*$ ,  $f(x) = M(x)$ . Then  $f$  is called a recursive function.

For example the function from  $\{0, 1\}^*$  to  $\{0, 1, \sqcup\}$ , that replaces 0 with 1 and vice versa until it encounters a  $\sqcup$ , is recursive by the example.

For ease of “programming” Turing-machines we define an obvious generalization.

**DEFINITION 9  $k$ -string Turing-machines**

A  $k$ -string Turing-machine is a quadruple  $M = (Q, \Sigma, \delta, s)$ , where  $Q$ ,  $\Sigma$  and  $s$  are exactly as in ordinary Turing-machines. But here  $\delta$  is a function from  $Q \times \Sigma^*$  to  $(Q \cup \{h, \text{“yes”}, \text{“no”}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$ . We still assume that  $\triangleright$  is at the start of each string, cannot be overwritten and upon reading it the cursor must move to the right.

The input of  $M$  is written on the first string and the output on the last string.

Intuitively,  $\delta$  decides the next state of the machine by looking at its current state and the symbol at the cursor of each band. It then overwrites the current symbol on each band by another and moves the cursors on each band either left or right or not at all.

Analogous to 1-string Turing-machines we define *configuration* and *yields* for  $k$ -string Turing-machines:

A *configuration* is a  $k + 2$  tuple, with the first coordinate being the current state, the even ones being the word on the left of the cursor, the odd ones the word on the right of the cursor.

*Yields* is defined exactly as in 1-string machines after taking the greater number of strings into account.

**Example 2 (Palindromes)** We now construct a 2-string Turing-machine, that decides the language of palindromes on  $\Sigma = \{0, 1\}$ . It first copies its input on the second string, then it moves the first cursor on the first symbol of the input and the second on the last symbol. Then it moves the two cursors in opposite direction comparing the two symbols at the cursors.

$q \in Q$	$l_1 \in \Sigma$	$l_2 \in \Sigma$	$\delta(q, l_1, l_2)$
$s$	$\triangleright$	$\triangleright$	$(s, \triangleright, \rightarrow, \triangleright, \rightarrow)$
$s$	$0$	$\sqcup$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s$	$1$	$\sqcup$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s$	$\sqcup$	$\sqcup$	$(q, \sqcup, \leftarrow, \sqcup, -)$
$q$	$0$	$\sqcup$	$(q, 0, \leftarrow, \sqcup, -)$
$q$	$1$	$\sqcup$	$(q, 1, \leftarrow, \sqcup, -)$
$q$	$\triangleright$	$\sqcup$	$(p, \triangleright, \rightarrow, \sqcup, \leftarrow)$
$p$	$0$	$0$	$(p, 0, \rightarrow, 0, \leftarrow)$
$p$	$1$	$1$	$(p, 1, \rightarrow, 1, \leftarrow)$
$p$	$0$	$1$	$(\text{"no"}, 0, \rightarrow, 1, \leftarrow)$
$p$	$1$	$0$	$(\text{"no"}, 1, \rightarrow, 0, \leftarrow)$
$p$	$\sqcup$	$\triangleright$	$(\text{"yes"}, \sqcup, \rightarrow, \triangleright, \leftarrow)$

There is no Turing-machine

significantly faster at deciding palindromes than this one.

## 1.2 Nondeterminism

So far we have examined “realistic” Turing-machines in the sense, that it is possible and feasible to build one; we now drop this notion and consider a different, “unrealistic” kind of Turing-machine.

### DEFINITION 10 Nondeterministic Turing-machines

A nondeterministic Turing-machine is a quadruple  $N = (Q, \Sigma, \delta, s)$ , where  $Q$ ,  $\Sigma$  and  $s$  are as in deterministic Turing-machines. But now  $\delta$  is a function from  $Q \times \Sigma$  to  $\mathcal{P}((Q \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\})$ . That is, for a nondeterministic machine there may be more than one appropriate next step or none at all. It can also be in multiple configurations at once.

A computation of a nondeterministic machine can be imagined as a computation tree where each path can be computed by a deterministic machine.

Analogous to deterministic machines we define additional concepts.

### DEFINITION 11 Nondeterministic yields

Similarly we define  $(q, w, u)$  yields  $(q', w', u')$  in one step as with deterministic machines except that only one  $(q', l', D) \in \delta(q, l)$  must match the definition of yields in one step for deterministic machines.

Yields in  $k$  steps and yields are defined exactly as with deterministic machines.

### DEFINITION 12 Nondeterministic decision of languages

We say  $N$  accepts a language  $L$ , if for any  $x \in \Sigma^*$  the following is true:  $x \in L$  if and only if  $(s, \triangleright, x)$  yields  $(\text{"yes"}, w, u)$  for some words  $w$  and  $u$ .

This definition of decision sets the nondeterministic machine apart from the deterministic one: An input is accepted if there is *one* computation path that

results in “yes”.

## 2 Complexity Classes

We now introduce now the notion of the time needed to compute an output.

### DEFINITION 13 $\text{TIME}(f(n))$

For an input  $x$  of a  $k$ -string Turing-machine  $M$  the time required by  $M$  on input  $x$  is simply the number of steps to halting. If  $M$  does not halt on  $x$  the time required is  $\infty$ .

Now let  $f : \mathbb{N} \rightarrow \mathbb{N}$  be a function. We say the  $M$  operates within time  $f(n)$  if, for any input  $x$ , the time required by  $M$  on  $x$  is at most  $f(|x|)$ .  $f$  is then called a time bound for  $M$ .

Finally let  $L \subset (\Sigma \setminus \{\sqcup\})^*$  be language decided by  $M$  operating in time  $f(n)$ . Then we say that  $L \in \text{TIME}(f(n))$ .

That is,  $\text{TIME}(f(n))$  contains exactly those languages that can be decided by a Turing-machine within time bound  $f(n)$ .

**Example 3 (Palindromes)** In our example we constructed a Turing-machine  $M$ , which decides the language of palindromes  $L$  on  $\{0, 1\}$ . Now let us count how many steps  $M$  needs in the *worst case* to accept a word  $x$  with length  $n$ .

At first the machine copies  $x$  to the second band and moves the first cursor to the first symbol needing  $2n + 3$  steps. Then it compares the symbols of the first band with the symbols on the second band needing  $n$  steps.

Altogether  $M$  needs at most  $3n + 3$  steps and as such  $L \in \text{TIME}(3n + 3)$ .

### 2.1 $P$ and $NP$

In this section we will define the most well known complexity classes:  $P$  and  $NP$ .

#### DEFINITION 14 $P$

$P$  is the set of all languages decidable by a Turing-machine in a polynomial time bound. That is:

$$P = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)$$

The number of strings does not significantly increase the speed of Turing-machines.

#### THEOREM 1 Simulating a $k$ -string Turing machine with 1 string

Given any  $k$ -string Turing machine  $M$  operating within time bound  $f(n)$ , there is a 1-string Turing-machine  $M'$  operating within time  $\mathcal{O}(f(n)^2)$ , such that  $M(x) = M'(x)$  for any input  $x$ .



PROOF We define  $M'$  by taking □

This theorem also holds for nondeterministic Turing-machines.

Now analogous to the definition of TIME and  $P$ :

**DEFINITION 15** **NTIME( $f(n)$ )**

We say that a nondeterministic Turing-machine  $N$  accepts a language  $L$  in time  $f(n)$ , where  $f : \mathbb{N} \rightarrow \mathbb{N}$ , if  $N$  decides  $L$  and for all words  $x$  with length  $n$  no computation path may be longer than  $f(n)$ .

Then  $\text{NTIME}(f(n))$  is the set of languages acceptable by a nondeterministic Turing-machine within time bound  $f(n)$ .

Note that we do not tax non-deterministic machines for the whole amount of computation going on; just for the longest path.

**DEFINITION 16** **NP**

$NP$  is the set of all languages acceptable by a nondeterministic Turing-machine within a polynomial time bound. That is:

$$NP = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$$

$NP$  does not depend on the number of strings of the Turing-machines.

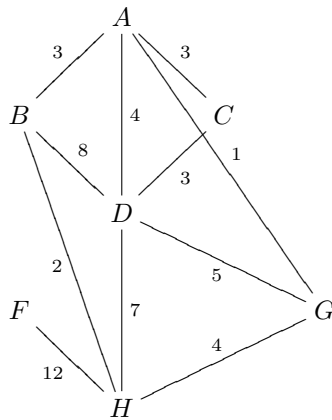
Obviously it holds that  $P \subset NP$ : The deterministic Turing-machines are exactly those nondeterministic machines where  $|\delta(q, l)| = 1$  for all  $q$  and  $l$ .

Whether the other inclusion also holds is unknown: This is the famous problem

$$P \stackrel{?}{=} NP.$$

It is strongly conjectured to be not the case.

**Example 4 (Traveling Salesman Problem)** Now an example of a problem and its discussion.



Given a weighted, possibly directed graph, what is the shortest path such that each vertex is visited at least once. It can be transformed in a decision problem: Given an integer  $B$  is there a path with length at most  $B$  such that each vertex is visited at least once. We call this problem  $\text{TSP}(D)$ .

By encoding the problem  $TSP(D)$  in an alphabet, we can use Turing-machines to solve it: the most obvious algorithm is calculating all routes and checking if the shortest one is shorter than  $B$  or not. If  $n$  is the number of vertices, then this takes about  $n!$  steps; the best known algorithm is not much better than that.

Still it is unknown whether there is an algorithm solving  $TSP(D)$  in polynomial time. If this were the case then  $P = NP$ .

But  $TSP(D)$  is obviously in  $NP$ : a nondeterministic Turing-machine can calculate all routes at once and check if it is shorter than  $B$  or not, a task which takes roughly  $n^2$  steps.

We now give a more intuitive characterization of  $NP$ .

Let  $R \subset \Sigma^* \times \Sigma^*$  be a relation.  $R$  is said to be *polynomially decidable* if there is a deterministic Turing-machine which decides the language  $\{x\#y, (x, y) \in R\}$  in polynomial time.  $R$  is called *polynomially balanced* if  $(x, y) \in R$  implies  $|y| \leq |x|^k$  for some  $k \geq 1$ .

**THEOREM 2** *Let  $L$  be a language. Then  $L \in NP$  if and only if there is a polynomially decidable and polynomially balanced relation  $R$ , such that  $L = \{x, (x, y) \in R \text{ for some } y\}$ .*

Such a  $y$  is called *certificate* for  $x$ . An example for a certificate would be the vertices of a path shorter than  $B$  for  $TSP(D)$ .

We can now describe  $P$  and  $NP$  informally:

$P$  contains those languages  $L$ , for which  $x \in L$  can be decided quickly

$NP$  contains those languages  $L$ , for which a certificate for  $x \in L$  can quickly be verified.

This is another reason why it is believed that  $P = NP$  does not hold: It is usually much easier to verify a solution than to find one.

**THEOREM 3** **Simulating a nondeterministic Turing-machine by a deterministic one**

*Let  $L$  be language decided by a nondeterministic Turing-machine  $N$  in time  $f(n)$ . Then there is deterministic Turing-machine  $M$  deciding  $L$  in time  $\mathcal{O}(c^{f(n)})$ , where  $c > 1$  depends on  $N$  alone.*

*Or put differently:*

$$NTIME(f(n)) \subset \bigcup_{c>1} TIME(c^{f(n)})$$

## 2.2 Space Complexity

We now introduce another way of quantifying “complexity”: by the sum of the maximum lengths of the strings.

To do this properly we have to introduce a special version of Turing-machines: Ones with special input and output bands. Otherwise we could not study machines which need asymptotically less or equal space for computation than the length of the input respectively output.

**DEFINITION 17  $k$ -string Turing-machine with input and output**

A (non)deterministic  $k$ -string Turing-machine,  $k \geq 3$ , with input and output is a Turing-machine that scans over its input only once, does not overwrite it and stops at the end of the input string. Furthermore it never moves the last cursor to the left.

**DEFINITION 18 Space required**

Let  $M$  be a  $k$ -string Turing-machine with input  $x$  such that the starting configuration  $(s, \triangleright, x, \triangleright, \epsilon, \dots)$  yields  $(H, w_1, u_1, \dots, w_k, u_k)$ , where  $H$  is halting state. Then the space required by  $M$  on input  $x$  is  $\sum_{i=1}^k |w_i u_i|$ .

If however  $M$  is a Turing-machine with input and output then the first and last band are not counted: the space required by  $M$  on input  $x$  is then  $\sum_{i=2}^{k-1} |w_i u_i|$ .

Now we can define the deterministic space complexity class just like the time complexity one.

**DEFINITION 19 SPACE( $f(n)$ )**

Let  $f : \mathbb{N} \rightarrow \mathbb{N}$  be a function and  $M$  be a Turing-machine. We say  $M$  operates within space bound  $f(n)$ , if for any input  $x$  the space required by  $M$  is at most  $f(|x|)$ .

We say a language  $L$  is in  $(f(n))$ , if there is a Turing-machine with input and output that decides  $L$  in space bound  $f(n)$ .

Now we define space complexity for nondeterministic machines.

**DEFINITION 20 Deciding in space  $f(n)$**

Let  $f : \mathbb{N} \rightarrow \mathbb{N}$  be a function and  $N$  be a  $k$ -string nondeterministic Turing-machine with input and output. We say  $N$  decides  $L$  in space  $f(n)$ , if  $N$  decides  $L$  and if for any input  $x$  this implication holds:

$$\begin{aligned} (s, \triangleright, x, \triangleright, \epsilon, \dots) &\xrightarrow{N^*} (q, w_1, u_1, \dots, w_k, u_k) \\ &\Rightarrow \\ &\sum_{i=2}^{k-1} |w_i u_i| \leq f(|x|) \end{aligned}$$

So  $N$  may not use in any computation path more space than  $f(|x|)$ . Note that we do not require for  $N$  to halt on all computations.

**DEFINITION 21**                    **NSPACE( $f(n)$ )**

Let  $f : \mathbb{N} \rightarrow \mathbb{N}$  be a function. We say that a language  $L$  is in  $\text{NSPACE}(f(n))$ , if there is a  $k$ -string nondeterministic Turing-machine with input and output that decides  $L$  in space  $f(n)$ .

Now completely analogous to  $P$  and  $NP$  for time we define for space complexity:

**DEFINITION 22**                    **Some space complexity classes**

$$PSPACE := \bigcup_{k=1}^{\infty} \text{SPACE}(n^k)$$

$$NPSPACE := \bigcup_{k=1}^{\infty} \text{NSPACE}(n^k)$$

**2.3 Relation between various Complexity Classes**

Now that we have seen quite a lot different complexity classes, we will clear some connections between them.

**THEOREM 4**                    **Relation between deterministic and nondeterministic classes**

- (i)  $\text{SPACE}(f(n)) \subset \text{NSPACE}(f(n))$ ,  $\text{TIME}(f(n)) \subset \text{NTIME}(f(n))$
- (ii)  $\text{NTIME}(f(n)) \subset \text{SPACE}(f(n))$
- (iii)  $\text{NSPACE}(f(n)) \subset \text{TIME}(k^{\log(n)} + f(n))$

For any complexity class  $S$  we can define  $\text{co}S = \{L, L^C \in S\}$  as the class of the complements of languages in  $S$ .

**THEOREM 5**                    **Deterministic classes and coclasses**

Any deterministic complexity class  $D$  is closed under complement, i.e. for  $L \in C$  its complement  $L^C$  is also in  $C$ .

**PROOF** Let  $L$  be a language in  $C$  and  $M$  be a Turing-machine that decides  $L$  within the bound required by the class  $D$ . Now let  $M'$  be the Turing-machine whose output is “no” whenever the output of  $M$  is “yes” and vice versa. Then  $M$  decides  $L^C$  in the same bound as  $M$  and therefore  $L^C \in D$ .                     $\square$

## THEOREM 6    Relation between space complexity classes

$$PSPACE = NPSPACE$$
$$NPSPACE = coNPSPACE$$

### Open questions

As we have seen, there are some unexpected relationships between the complexity classes, but for some of the most obvious questions it remains unknown whether:

- $P = NP$ ?
- $NP = coNP$ ?

Both answers are suspected to be “no”, but so far a proof supporting either way remains elusive since the beginning of the theory of complexity classes.

Various deterministic and nondeterministic complexity classes can be defined analogous to  $P$  and  $NP$  using other functions than polynomials, e.g. logarithms or exponentials.

For all of these it is also unknown whether they are equal or not.

## 2.4 Completeness

If we want to prove that some language is in a complexity class, it is cumbersome to start from scratch every time; one wants to use something already known to shorten the process. One of the most useful techniques for this is *reduction*.

### DEFINITION 23    Reduction

Let  $\Sigma$  and  $\Gamma$  be alphabets and  $L_1 \subset \Sigma^*$  and  $L_2 \subset \Gamma^*$  be languages. We say that  $L_1$  is reducible to  $L_2$ , if there is a function  $f : \Sigma^* \rightarrow \Gamma^*$  computable by a deterministic Turing-machine in polynomial time such that the following holds for all inputs  $x$ :  $x \in L_1$  if and only if  $f(x) \in L_2$ .  $f$  is then called reduction from  $L_1$  to  $L_2$ .

Note that the composition of reductions is also a reduction; it follows that *reducible* is a transitive relation.

And here is the reason, why reductions are useful to shorten proofs:

THEOREM 7  $P, NP, coNP, PSPACE, NPSPACE$  are closed under reductions, i.e. if  $L_2 \in P/NP/coNP/PSPACE/NPSPACE$  and  $L_1$  can be reduced to  $L_2$  then  $L_1 \in P/NP/coNP/PSPACE/NPSPACE$ .

There are many more classes that are closed under reductions; in fact one expects for a sensible complexity class to be closed under some kind of reduction.

**DEFINITION 24                      Completeness**

Let  $L$  be a language in a complexity class  $C$ . Then  $L$  is called  $C$ -complete if any language  $L' \in C$  can be reduced to  $L$ .

Reduction and completeness can be very useful tools for establishing relations between the classes:

**THEOREM 8** *Let  $C$  be a complexity class and  $D \subset C$  be another complexity class closed under reductions. If  $L \in C$  is a  $C$ -complete language, then  $L \in D$  if and only if  $C = D$ .*

**PROOF** “ $\Leftarrow$ ”: trivial. “ $\Rightarrow$ ”: Let  $L'$  a language in  $C$ ; since  $L \in C$  is complete,  $L'$  can be reduced to  $L$ . The reduction is also in  $D$  since  $D$  is closed under reductions. Therefore  $C = D$ .  $\square$

 **$NP$ -complete problems**

So one way to solve  $P \stackrel{?}{=} NP$  or similar problems positively, is to search for a  $NP$ -complete problem and find a polynomial time deterministic algorithm to solve it.

Here is short list of such problems:

- SAT
- HAMILTON PATH
- TSP(D)
- CLIQUE
- KNAPSACK
- BIN PACKING
- PARTITION
- ...

## Part II

# Propositional Logic

As we will mainly discuss the complexity of propositional proof systems we shall now introduce basics of *propositional logic* and formally describe “proofs” in this logic using Turing-machines.

### DEFINITION 1            **Syntax of propositional logic**

Let  $V = \{x_1, x_2, \dots\}$  be a countable set of variables and  $C$  be a finite set of logical connectives with arbitrary, finite arity. Then all of the following are propositional formulas:

- The signs  $\perp$  and  $\top$
- Variables
- $\oplus(p_1, \dots, p_k)$ , where  $\oplus \in C$  is a logical connective (with arity  $k$ ) and  $p_1, \dots, p_k$  are propositional formulas.

Parentheses may be added to clarify the precedence of logical connectives.

### DEFINITION 2            **Valuation**

A valuation  $v : V \rightarrow \{\top, \perp\}$  is a function that gives variables the values “ $\top$ ” or “ $\perp$ ”

### DEFINITION 3            **Propositional language**

Given a set of variables  $V$ , a finite set of connectives  $C$  and a valuation  $v$  a propositional language defines the value of a propositional formula inductively:

- $\top$  means “true”,  $\perp$  means “false”
- Each  $x \in V$  means  $v(x)$
- For each  $k \in \mathbb{N}_0$ , each connective  $\oplus$  in  $C$  and each  $k$ -tuple of formulas  $(t_1, \dots, t_k)$  each  $t_i$  meaning either “true” or “false” it defines whether  $\oplus(t_1, \dots, t_k)$  is “true” or “false”

After fixing a language we can give any formula  $\phi$  a meaning of either “true” or “false” using a valuation  $v$ . For this we write  $\phi(v)$ .

**Example 5 (A propositional logic)** Here is a well known propositional language:

$$V = \{x_1, \dots\}, C = \{\vee, \wedge, \neg\}$$

Examples of such propositional formulas

$$x_1 \wedge x_2, (\neg(x_3 \vee x_1)) \wedge x_1, \dots$$

With the usual meaning of the connectives this propositional language is called *standard base*. Usually when we talk about propositional formulas, we mean this language.

The other well known symbols  $p \Rightarrow q$  and  $p \Leftrightarrow q$  can be seen as abbreviations of  $\neg p \vee (p \wedge q)$  or  $(p \Rightarrow q) \wedge (q \Rightarrow p)$ .

The standard way to prove something about a propositional formula  $\phi$  is the *induction on the structure of  $p$* . For example:

**THEOREM 1** *Let  $\phi$  be a propositional formula and  $v_1$  and  $v_2$  be valuations that agree on the set  $P$  of variables in  $\phi$ , i.e.  $v_1|_P = v_2|_P$ . Then  $\phi(v_1) = \phi(v_2)$ .*

**PROOF** Proof by induction on the structure of  $\phi$ .

- $\phi = \perp$  or  $\phi = \top$ :  
 $\phi(v) = \text{“true”}$  respectively “false” regardless of valuation  $v$
- $\phi = x$  with  $x$  variable:  
 $\phi(v_1) = v_1(x) = v_2(x) = \phi(v_2)$  because  $v_1$  and  $v_2$  agree on  $\{x\}$ .
- $\phi = \oplus(\tau_1, \dots, \tau_k)$  where  $\tau_i$  are propositional formulas for which the hypothesis  $\tau_i(v_1) = \tau_i(v_2)$  holds:  
 $\phi(v_1) = \oplus(\tau_1(v_1), \dots, \tau_k(v_1)) = \oplus(\tau_1(v_2), \dots, \tau_k(v_2)) = \phi(v_2)$

Therefore the hypothesis holds. □

Many proofs about propositional formulas follow the same scheme.

#### DEFINITION 4      **Satisfiability and Tautology**

*A propositional formula  $\phi$  is called*

**satisfiable** , *if there is a valuation  $v$  such that  $\phi(v) = \text{“true”}$ .*

**unsatisfiable** , *if there is no such valuation.*

**tautology** , *if for each valuation  $v$   $\phi(v) = \text{“true”}$ .*

**equivalent** *to a formula  $\psi$  if  $\phi \Leftrightarrow \psi$  is a tautology.*

*The set of all satisfiable formulas is called SAT, the set of all tautology TAUT.*

#### About SAT and TAUT

SAT is in *NP*: A nondeterministic machine can “guess” all valuations of a formula at once and then check if one is true. This can be done polynomial time.

It even is *NP*-complete: For any nondeterministic machine  $N$  deciding  $L$  in polynomial time and any input  $x$ , there is a formula  $F$  constructible in logarithmic space that is satisfiable if and only if  $x \in L$ .

Instead of asking if  $\phi \in \text{TAUT}$  one can ask if  $\neg\phi \in \text{SAT}$ . In that sense SAT and TAUT are essentially the same.



### 3 Abstract Proof Systems

**DEFINITION 5 (Abstract) Proof systems**

A proof system for a language  $L$  is a deterministic Turing-machine  $M$  operating in polynomial time that for any word  $x$ , there exists a certificate  $p$ , such that  $M$  accepts  $x\#p$  if and only if  $x \in L$ .

Recall the characterization of  $NP$  through certificates: It is clear that the language  $L$  is in  $NP$ .

In that case,  $p$  is called a *proof* for  $x$ .

The time bound of the machine  $M$  is called (*time*) *complexity* of the proof.

**DEFINITION 6 Propositional proof system**

A propositional proof system is a proof system for the language  $TAUT$  of tautologies.

If there exists a proof  $p$  in this system for every tautology  $x$ , then the system is called *complete*.

If the existence of a proof  $p$  for  $x$  in this system implies that  $x$  is a tautology, then the system is called *sound*.

Since a formula is a tautology if and only if its negation is unsatisfiable, we can give an alternative definition:

**DEFINITION 7 Propositional proof system**

A propositional proof system is a proof system for the language  $UNSAT$  of unsatisfiable formulas.

Since there is a great number of propositional proof systems, it is useful to have a way of saying that one is as strong as another.

**DEFINITION 8 Simulation**

Let  $M$  and  $M'$  be two proof systems. We say that  $M$  polynomially simulates  $M'$  if  $M$  and  $M'$  prove the same language and proofs in  $M'$  can be polynomially converted into proofs in  $M$ , i.e. there exists a in polynomial time computable function such that  $p$  is a proof in  $M'$  if and only if  $f(p)$  is a proof in  $M$ .

**DEFINITION 9 Equivalence of proof systems**

Two proof systems  $M$  and  $M'$  are said to be equivalent if  $M$  polynomially simulates  $M'$  and vice versa.

## 4 Example of a Propositional Proof System: Resolution

Now a short introduction to a common proof system: *Resolution*

The proof systems will be described algorithmically rather than stating the corresponding Turing machine.

### Resolution

#### DEFINITION 10 **Literal, Conjunctive and Disjunctive Normal Form**

A propositional formula  $\phi$  is

**a literal** if  $\phi$  is a variable or a negated variable.

**in conjunctive normal form (CNF)** if it is a conjunction of disjunctions of literals.

**in disjunctive normal form (DNF)** if it is a disjunction of conjunctions of literals.

Any formula has an equivalent formula in CNF and one in DNF.

#### DEFINITION 11 **Equisatisfiable**

Two formulas  $\phi$  and  $\phi'$  are called *equisatisfiable*, if  $\phi$  is satisfiable if and only if  $\phi'$  is satisfiable.

**Example 6**  $(x \Leftrightarrow p) \wedge (x \vee q)$  and  $p \vee q$ , with  $p, q$  formulas, are equisatisfiable, but not equivalent.

Let  $\phi$  be a formula in CNF represented as a set  $\Phi$  of sets of literals, e.g.  $\phi = (x_1 \vee x_2) \wedge x_3 \wedge \neg x_2$  is represented as  $\Phi = \{\{x_1, x_2\}, \{x_3\}, \{\neg x_2\}\}$ . *Resolution* is an algorithm for deciding if  $\phi \in \text{SAT}$  working on that set  $\Phi$ .

**Example 7** Let us now see how resolution might be used to proof or disproof the satisfiability of a formula:

$$\begin{aligned} & \{\{a, b\}, \{a, \neg b\}, \{\neg a, b\}, \{\neg a, \neg b\}\} \\ & \quad \{\{a\}, \{\neg a\}\} \\ & \quad \quad \emptyset \end{aligned}$$

Each step in this example *resolved on* a variable: On the first we resolved on  $b$  using two resolution steps, on the second on  $a$ .

The formula represented by this set of sets is not satisfiable.

**DEFINITION 12                      Resolvent**

Let  $C$  and  $D$  be set of literals representing a disjunction and  $x$  be variable. If  $x \in C$  and  $\neg x \in D$ , then the set  $(C \setminus \{x\}) \cup (D \setminus \{\neg x\})$  is called the resolvent of  $C$  and  $D$ .

Taking two clauses and replacing them by their resolvent is called one *resolution step*

**DavisPutnam algorithm**

Now we will build from a set of clauses all possible resolvents replacing the original clauses. Let  $S$  be a set of sets of literals representing a CNF and  $l$  be a variable. Define

- $P := \{C \setminus \{l\}, l \in C \in S\}$
- $N := \{D \setminus \{\neg l\}, \neg l \in D \in S\}$
- $S_0 := \{E \in S, l, \neg l \notin E\}$
- $S' := S_0 \cup \{C \cup D, C \in P, D \in N\}$

Then the formulas represented by  $S$  and  $S'$  are equisatisfiable. Building  $S'$  from  $S$  is one step in the DavisPutnam algorithm.

Let  $S$  be a set of sets of literals. Construct  $S'$  by one DavisPutnam step from  $S$ ,  $S''$  by one resolution step from  $S'$ ,  $S^{(3)}$  from  $S''$  and so on, each time using a new variable.

If there is a  $n \in \mathbb{N}$  such that  $\emptyset \in S^{(n)}$  then the formula represented by  $S^{(n)}$  is unsatisfiable and therefore also the one represented by  $S$ .

Is it not possible to choose a variable to construct a new resolvent anymore, then the formula represented by  $S$  is satisfiable.

Resolution is therefore a complete and sound proof system for propositional formulas in CNF.

## References

- [1] S. Buss. An introduction to proof theory. *Handbook of Proof Theory*, pages 1–78, 1998.
- [2] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [3] P. Pudlak. On the complexity of the propositional calculus. *Sets and Proofs: Invited papers from Logic Colloquium'97*, pages 197–218, 1999.